Binary Decision Diagrams and Their Applications for VLSI CAD(Dissertation_全文)

AUTHOR(S):

Minato, Shin-ichi

(2)

Binary Decision Diagrams and Their Applications for VLSI CAD

Shin-ichi Minato

December 1994

Abstract

Manipulation of Boolean functions is a fundamental of computer science. Many problems in digital system design and testing can be expressed as a sequence of operations on Boolean functions. With the recent advance in very large-scale integration (VLSI) technology, the problems grow large beyond the scope of manual design, and the computer-aided design (CAD) systems have become widely used. The performances of these systems greatly depend on the efficiency of Boolean function manipulation. It is a very important technique not only in VLSI CAD systems but also in many problems of computer science, such as artificial intelligence and combinatorics.

A key to efficient Boolean function manipulation is to have a good data structure. *Binary Decision Diagrams (BDDs)* are graph representations of Boolean functions. The basic concept was introduced by Akers in 1978, and efficient manipulation methods was developed by Bryant in 1986. Since then, BDDs have attracted the attention of many researchers because of their good properties to represent Boolean functions. A BDD gives a canonical form for a Boolean function, so that we can easily check the equivalence of two functions. Although a BDD may become exponential size for the number of inputs in the worst case, the size varies with the kind of functions, unlike truth table always require 2^n bit of memory. It is known that many practical functions can be represented by a feasible size of BDDs. This is an attractive feature of BDDs.

This thesis discusses the techniques related to BDDs and their applications for VLSI CAD systems. In Chapter 2, we start with describing the basic concept of BDDs and Shared BDDs. We then present the algorithms of Boolean function manipulation using BDDs. In implementing BDD manipulators on computers, the memory management is an important issue for the system performance. We show such implementation techniques to make BDD manipulators applicable to practical problems. As an improvement of BDDs, we propose the use of *attributed edges*, which are the edges attached with several sorts of attributes representing a certain operation. Using these techniques, we implemented a BDD subroutine package for Boolean function manipulation. It can efficiently represent and manipulate very large-scale BDDs containing more than million of nodes. Such Boolean functions have never been dealt with by other classical methods. We show some experimental results to evaluate the applicability of the BDD package to practical problems.

In Chapter 3, We discuss the variable ordering for BDDs. It is important for

utilizing BDDs since the size of BDDs greatly depends on the order of the input variables. It is difficult to derive a method that always yields the best order to minimize BDDs, but with some heuristic methods, we can find a fairly good order in many cases. We first consider general properties on variable ordering for BDDs. Based on the consideration, we propose two heuristic methods of variable ordering. One method, named *dynamic weight assignment method*, finds an appropriate order before generating the BDD. It refers topological information of the Boolean expression or logic circuit which specifies the sequence of BDD operations. The other method, named *minimum-width method*, reduces BDD size by reordering the input variables for a given BDD with a certain initial order. We implemented the two methods and conducted some experiments. Experimental results shows that our methods are effective to reduce BDD size in many cases, and useful for practical applications.

In Chapter 4, We discuss the representation of multi-valued functions. In many problems in digital system design, we sometimes use ternary-valued functions containing don't cares. There are two methods to extend BDDs to deal with ternary logics; *ternary-valued BDDs* and using *a pair of BDDs*. We compare and clarify the relationship of the two methods by introducing a special input variable, called *D-variable*. In this discussion, we show that the difference of the two method can be concluded into variable ordering. This argument is extended into n-ary-valued functions. Some variant of BDDs have been devised to represent multi-valued functions. We survey these methods and compare them as well as on the ternary-valued functions.

One other topic is how efficiently transform BDD representation into other data structures. Chapter 5 presents a fast method for generating prime-irredundant forms of cube sets from given BDDs. Prime-irredundant means a form such that each cube is a prime implicant and no cube can be eliminated. Our algorithm generates compact cube sets directly from BDDs, in contrast to the conventional cube set reduction algorithms, which commonly manipulate redundant cube sets or truth tables. Our method is based on the idea of a *recursive operator*, proposed by Morreale. Morreale's algorithm is also based on cube set manipulation. We found that the algorithm can be improved and rearranged to fit BDD operations efficiently. The experimental results demonstrate that our method is efficient in terms of time and space. In practical time, we can generate cube sets consisting of more than 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method is more than 10 times faster than conventional methods in large-scale examples. It gives quasi-minimum numbers of cubes and literals. This method will find many useful applications in logic design systems.

As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* to deal with many problems, not only in the digital system design but also various areas in computer science. By mapping a set of combinations into the Boolean space, it can be represented as a characteristic function using a BDD. This method enables us to manipulate a huge number of combinations

implicitly, which has never been practical before. However, this BDD-based set representation does not completely match the properties of BDDs, therefore sometimes the size of BDDs grow large because the reduction rules are not effective. There is room to improve the data structure for representing sets of combinations.

In Chapter 6, we propose *Zero-Suppressed BDDs (0-Sup-BDDs)*, which are BDDs based on a new reduction rule. This data structure is adapted to represent *sets of combinations*. 0-sup-BDDs can manipulate sets of combinations more efficiently than using conventional BDDs. We discuss the properties of 0-sup-BDDs and their efficiency based on a statistical experiment. We then present the basic operators for 0-sup-BDDs. Those operators are defined as the operations on sets of combinations, which slightly differ from the Boolean function manipulation based on conventional BDDs.

When describing algorithms or procedures for manipulating BDDs, we usually use Boolean expressions based on switching algebra. Similarly, when considering sets of combinations with 0-sup-BDDs, we can use *unate cube set* expressions and their algebra. Based on unate cube set algebra, we can simply describe algorithms or procedures for 0-sup-BDDs. We developed efficient algorithms for executing unate cube set operations including multiplication and division. Here we discuss calculation of unate cube set algebra using 0-sup-BDDs. We propose efficient algorithms for computing unate cube set operations, and show some practical applications.

In Chapter 7, an application for VLSI logic synthesis is presented. We propose a fast factorization method for cube set representation represented with 0-sup-BDDs. Our new algorithm can be executed in a time almost proportional to the size of 0-sup-BDDs, which are usually much smaller than the number of literals in the cube set. By using this method, we can quickly generate multi-level logics from implicit cube sets even for parity functions and full-adders, which have never been possible with the conventional methods. We implemented a new multi-level logic synthesizer, and experimental results indicate our method is much faster than conventional methods and differences are more significant for larger-scale problems. Our method greatly accelerates multi-level logic synthesis systems and enlarges the scale of applicable circuits.

In Chapter 8, we presents a helpful tool for the research on computer science. When we are considering problems related to logics, we sometimes faced with the task to describe and calculate Boolean expressions. It is a cumbersome job to calculate or reduce Boolean expressions by hand, so we developed a computer-aided Boolean expression manipulator. Our product, called *BEM-II*, features that it calculates not only binary logic operation but also arithmetic operations on multi-valued logics, such as addition, subtraction, multiplication, division, equality and inequality. Such arithmetic operations provide simple descriptions for various problems. BEM-II feeds and computes the problems represented by a set of equalities and inequalities, which are dealt with using 0-1 linear programming. We discuss the data structure and algorithms for the arithmetic operations. Finally we present the specification of BEM-II and some application

examples, such as the 8-Queens problem. Experimental results indicate that it has a good computation performance in terms of the total time for programming and execution.

Contents

Chapter 1

Introduction

1.1 Background

Manipulation of Boolean functions is a fundamental of computer science. Many problems in digital system design and testing can be expressed as a sequence of operations on Boolean functions. With the recent advance in very large-scale integration (VLSI) technology, the problems grow large beyond the scope of manual design, and the computer-aided design (CAD) systems have become widely used. The performances of these systems greatly depend on the efficiency of Boolean function manipulation. It is a very important technique not only in VLSI CAD systems but also in many problems of computer science, such as artificial intelligence and combinatorics.

A key to efficient Boolean function manipulation is to have a good data structure. It is required to perform the following basic tasks efficiently in terms of execution time and memory space.

- Generating a Boolean function data which is the result of a logic operation, such as AND, OR, NOT, and EXOR, for given Boolean functions.
- Checking tautology or satisfiability of a given Boolean function.
- Finding an assignment of input variables such that a given Boolean function become 1, or counting the number of such assignments.

Various methods have been developed for representing and manipulating Boolean functions. There are several classical methods, such as *truth tables, parse trees* and *cube sets.*

Truth tables are suitable for manipulating on computers, especially on recent high-speed vector processors[IYY87] or parallel machines. However, they need 2^n bits of the memory to represent an n-input function, even for very simple functions. For example, a 100-input tautology function requires a 2^{100} bit of truth table. Since exponential memory requirement leads to an exponential computation time, truth tables are impractical for manipulating Boolean functions with many input variables.

Parse trees for Boolean expressions sometimes give compact representations for the functions with many input variables, which cannot be represented compactly using truth tables. However, there exist many different expressions for a given function. The equivalence checking of the two expressions is very hard as it is an NP problem, although there have been developed the method of rule-based transformation of the Boolean expressions[LCM89].

Cube sets (also called sum-of-products, PLA forms, covers, or two-level logics) are regarded as a special form of the Boolean expressions with the AND-OR two level structure. They have been extensively studied for many years and employed to represent Boolean function on computers. Cube sets sometimes give more compact representation than truth tables; however, redundant cubes may appear in logic operations, so they have to be reduced to check tautology or equivalency. This reduction process is time consuming. There are other drawbacks that NOT operation cannot be performed easily, and that parity functions become exponential sizes.

Unfortunately, the above methods are impractical for large scale problems because of their drawbacks. An efficient method for representing practical Boolean functions have been desired.

Binary Decision Diagrams (BDDs) are graph representations of Boolean functions. The basic concept was introduced by Akers in 1978[Ake78], and efficient manipulation methods was developed by Bryant in 1986[Bry86]. Since then, BDDs have attracted the attention of many researchers because of their good properties to represent Boolean functions. A BDD gives a canonical form for a Boolean function, so that we can easily check the equivalence of two functions. Although a BDD may become exponential size for the number of inputs in the worst case, the size varies with the kind of functions, unlike truth table always require 2^n bit of memory. It is known that many practical functions can be represented by a feasible size of BDDs. This is an attractive feature of BDDs.

There have been a number of attempts to improve the BDD technique in terms of execution time and memory space. One of them is the technique of *shared BDDs (SBDDs)*[MIY90], or multi-rooted BDDs, which manage a set of BDDs by joining them into a single graph. This method reduces memory requirement and makes easy to check the equivalence of two BDDs. Another improvement of BDDs is the *negative edges*, or typed edges[MB88] They are attributed edges such that each edge has an information of inverting. They are effective to reduce the operation time and the size of the graph.

Using BDDs with those improvement methods, Boolean function manipulators are implemented on workstations and now widely distributed as BDD packages[Min90]. They have been tried to utilize in various applications, especially in the VLSI CAD systems, such as formal verification[FFK88, MB88, BCMD90], logic synthesis[CMF93, MSB93], and testing[CHJ+90, TIY91].

BDDs have excellent properties to manipulate Boolean functions; however, there are some problems to be considered when utilizing BDDs to practical applications. One of the problems is variable ordering. Conventional BDDs requires to fix the order of input variables, and the size of BDDs greatly depends

on the order. It is hard to find the best order which minimize BDDs. Variable ordering algorithm is one of the most important issues for utilizing BDDs. As another problem, we sometimes manipulate ternary valued functions containing *don't care* to mask unnecessary information. In such cases, we have to devise a way of representing don't cares since usual BDDs deals with only binary logics. This issue can be generalized into the method of manipulating multi-valued logics or integer functions using BDDs. One other topic is how efficiently transform BDD representation into other data structures, such as cube sets, or Boolean expressions. This method is important in practical applications to output the result of BDD manipulation.

As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* in many problems. One proposal is for multiple fault simulation by representing sets of fault combinations with BDDs[TIY91]. Two others are verification of sequential machines using BDD representation for state sets[BCMD90], and computation of prime implicants using *Meta Products*[CMF93], which represent cube sets using BDDs. There is also general method for solving binate covering problems using BDDs[LS90]. By mapping a set of combinations into the Boolean space, it can be represented as a characteristic function using a BDD. This method enables us to manipulate a huge number of combinations implicitly, which has never been practical before. However, this BDD-based set representation does not completely match the properties of BDDs, therefore sometimes the size of BDDs grow large because the reduction rules are not effective. There is room to improve the data structure for representing sets of combinations.

1.2 Outline of the Thesis

This thesis discusses the techniques related to BDDs and their applications for VLSI CAD systems. Chapter 2 to 5 discuss implementation and utility techniques of BDDs. Chapter 6 and 7 propose *zero-suppressed BDDs*, which is a variant of BDD adapted for representing sets of combinations. Chapter 8 presents an arithmetic Boolean expression manipulator, which is a helpful tool to the research on computer science.

In Chapter 2, we start with describing the basic concept of BDDs and Shared BDDs. We then present the algorithms of Boolean function manipulation using BDDs. In implementing BDD manipulators on computers, the memory management is an important issue for the system performance. We show such implementation techniques to make BDD manipulators applicable to practical problems. As an improvement of BDDs, we propose the use of *attributed edges*, which are the edges attached with several sorts of attributes representing a certain operation. They are regarded as a generalization of the *inverter*[Ake78], or typed edges[MB88]. Using these techniques, we implemented a BDD subroutine package for Boolean function manipulation. It can efficiently represent and

manipulate very large-scale BDDs containing more than million of nodes. Such Boolean functions have never been dealt with by other classical methods. We show some experimental results to evaluate the applicability of the BDD package to practical problems.

In Chapter 3, We discuss the variable ordering for BDDs. It is important for utilizing BDDs since the size of BDDs greatly depends on the order of the input variables[FFK88]. It is difficult to derive a method that always yields the best order to minimize BDDs, but with some heuristic methods, we can find a fairly good order in many cases. We first consider general properties on variable ordering for BDDs. Based on the consideration, we propose two heuristic methods of variable ordering. One method, named *dynamic weight assignment method*, finds an appropriate order before generating the BDD. It refers topological information of the Boolean expression or logic circuit which specifies the sequence of BDD operations. The other method, named *minimum-width method*, reduces BDD size by reordering the input variables for a given BDD with a certain initial order. We implemented the two methods and conducted some experiments. Experimental results shows that our methods are effective to reduce BDD size in many cases, and useful for practical applications.

In Chapter 4, We discuss the representation of multi-valued logic functions. In many problems in digital system design, we sometimes use ternary-valued functions containing don't cares. There are two methods to extend BDDs to deal with ternary-valued logics; *ternary-valued BDDs* and using *BDD pairs*. We compare and clarify the relationship of the two methods by introducing a special input variable, called *D-variable*. In this discussion, we show that the difference of the two method can be concluded into variable ordering. This argument is extended into multi-valued logic functions which deal with integer values. Some variants of BDDs have been devised to represent multi-valued logic functions, such as *multi-terminal BDDs (MTBDDs)*[BFG+93] and using *BDD vectors* which we proposed[Min93a]. We describe these methods and compare them as well as on the ternary-valued functions.

Chapter 5 presents a fast method for generating prime-irredundant forms of cube sets from given BDDs. Prime-irredundant means a form such that each cube is a prime implicant and no cube can be eliminated. Our algorithm generates compact cube sets directly from BDDs, in contrast to the conventional cube set reduction algorithms, which commonly manipulate redundant cube sets or truth tables. Our method is based on the idea of a *recursive operator*, proposed by Morreale. Morreale's algorithm is also based on cube set manipulation. We found that the algorithm can be improved and rearranged to fit BDD operations efficiently. The experimental results demonstrate that our method is efficient in terms of time and space. In practical time, we can generate cube sets consisting of more than 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method is more than 10 times faster than *ESPRESSO*[BHMSV84] in large-scale examples. It gives quasi-minimum numbers of cubes and literals. This method will find many useful applications in logic design systems.

In Chapter 6, we propose *Zero-Suppressed BDDs (0-Sup-BDDs)*, which are BDDs based on a new reduction rule. This data structure is adapted to *sets of combinations*, which appear in many combinatorial problems. 0-sup-BDDs can manipulate sets of combinations more efficiently than using conventional BDDs. We discuss the properties of 0-sup-BDDs and their efficiency based on a statistical experiment. We then present the basic operators for 0-sup-BDDs. Those operators are defined as the operations on sets of combinations, which slightly differ from the Boolean function manipulation based on conventional BDDs.

When describing algorithms or procedures for manipulating BDDs, we usually use Boolean expressions based on switching algebra. Similarly, when considering sets of combinations with 0-sup-BDDs, we can use *unate cube set* expressions and their algebra. Based on unate cube set algebra, we can simply describe algorithms or procedures for 0-sup-BDDs. We developed efficient algorithms for executing unate cube set operations including multiplication and division. Here we discuss calculation of unate cube set algebra using 0-sup-BDDs. We propose efficient algorithms for computing unate cube set operations, and show some practical applications.

In Chapter 7, an application for VLSI logic synthesis is presented. We propose a fast factorization method for cube set representation represented with 0-sup-BDDs. Our new algorithm can be executed in a time almost proportional to the size of 0-sup-BDDs, which are usually much smaller than the number of literals in the cube set. By using this method, we can quickly generate multi-level logics from implicit cube sets even for parity functions and full-adders, which have never been possible with the conventional methods. We implemented a new multi-level logic synthesizer, and experimental results indicate our method is much faster than conventional methods and differences are more significant for larger-scale problems. Our method greatly accelerates multi-level logic synthesis systems and enlarges the scale of applicable circuits.

In Chapter 8, we presents a helpful tool for the research on computer science. When we are considering problems related to logics, we sometimes faced with the task to describe and calculate Boolean expressions. It is a cumbersome job to calculate or reduce Boolean expressions by hand, so we developed a computer-aided Boolean expression manipulator. Our product, called *BEM-II*, features that it calculates not only binary logic operation but also arithmetic operations on multi-valued logics, such as addition, subtraction, multiplication, division, equality and inequality. Such arithmetic operations provide simple descriptions for various problems. BEM-II feeds and computes the problems represented by a set of equalities and inequalities, which are dealt with using 0-1 linear programming. We discuss the data structure and algorithms for the arithmetic operations. Finally we present the specification of BEM-II and some application examples, such as the 8-Queens problem. Experimental results indicate that it has a good computation performance in terms of the total time for programming and execution.

In Chapter 9, the conclusion of this thesis and future works are stated.

Chapter 2

Techniques of BDD Manipulation

2.1 Introduction

This section introduces basic definition of BDDs and shared BDDs which will be discussed in this thesis.

2.1.1 BDDs

BDDs are graph representation of Boolean functions, as shown Fig. 2.1(a). The basic concept was introduced by Akers[Ake78], and an efficient manipulation method was developed by Bryant[Bry86].

A *BDD* is a directed acyclic graph with two terminal nodes, which we call the *0-terminal node* and *1-terminal node*. Every non-terminal node has an index to identify an input variable of the Boolean function, and has two outgoing edges, called the *0-edge* and *1-edge*.

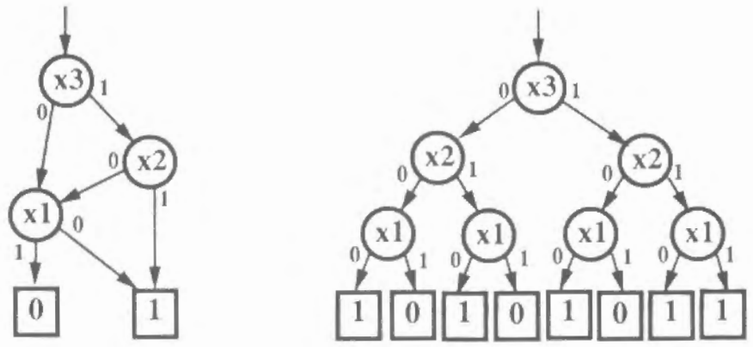
An *Ordered BDD (OBDD)* is a BDD such that the input variables appear in a fixed order in all the paths of the graph, and that no variable appears more than once in a path. In this thesis, we use natural number $1, 2, \ldots$ for the indexes of the input variables, and every non-terminal node has a greater index than its descendant nodes.

A compact OBDD is derived by reducing a binary tree graph, as shown in Fig. 2.1(b). In the binary tree, 0-terminals and 1-terminals represent logic values (0/1), and each node represents the *Shannon's expansion* of the Boolean function:

$$f = \overline{x_i} \cdot f_0 \vee x_i \cdot f_1,$$

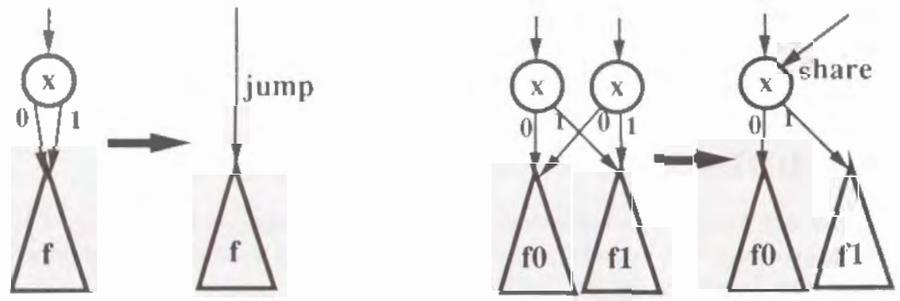
where i is the index of the node. f_0 and f_1 are the functions of the nodes pointed by 0- and 1-edges, respectively.

The following reduction rules give a *Reduced Ordered BDD (ROBDD)*.



(a) A BDD for $(x3 \cdot x2 \vee x1)$. (b) A binary decision tree.

Figure 2.1: A BDD and a binary decision tree.



(a) Node elimination. (b) Node sharing.

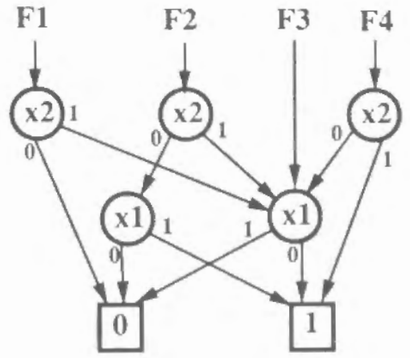
Figure 2.2: Reduction rules of BDDs.

1. Eliminate all the redundant nodes whose two edges point to the same node. (Fig. 2.2(a))

2. Share all the equivalent sub-graphs.(Fig. 2.2(b))

ROBDDs give canonical forms for Boolean functions when the variable order is fixed. This property is very important to practical applications, as we can easily check the equivalence of two Boolean function by only checking isomorphism of their ROBDDs. Most works about BDDs are based on the technique of the ROBDDs. In this thesis, we refer to ROBDDs as BDDs for the sake of simplification.

Since there are 2^{2^n} kinds of n-input Boolean functions, the representation requires at least 2^n bit of memory in the worst case. It is known that a BDD for an n-input function includes $O(2^n/n)$ nodes in general[LL92]. As each node consumes about $O(n)$ bit (to distinguish the two child nodes from $O(2^n/n)$



$(F1 = x2 \cdot \overline{x1},\ F2 = x2 \oplus x1,\ F3 = \overline{x1},\ F4 = x2 \vee \overline{x1}.)$

Figure 2.3: A shared BDD.

nodes), the total storage exceeds 2^n bit. However, the size of BDDs varies with the kind of function, unlike truth tables which always require 2^n bit of memory. There is a class of Boolean functions that can be represented by a polynomial size of BDDs, and many practical functions fall into this class[IY90]. This is an attractive feature of BDDs.

2.1.2 Shared BDDs

A set of BDDs representing multiple functions can be united into a graph which consists of BDDs sharing their sub-graphs with each other, as shown in Fig. 2.3. We call such graphs *Shared BDDs (SBDDs)*[MIY90], or *multi-rooted BDDs*. Two isomorphic sub-graphs cannot coexist in an SBDD. The following advantages are obtained by managing all the BDDs in a single graph.

- After generating BDDs, the equivalence of two functions can be checked in a constant time, by only referring the pointers to the root nodes.

- We can save the time and memory space to duplicate BDDs by only copying a pointer to the root node.

On the manipulation algorithms of BDDs, the original method for non-shared BDDs was presented by Brayant[Bry86]. However, shared BDDs are now widely used and their algorithms are simpler than the Bryant's method. In this thesis, we discuss on the methods based on the shared BDD techniques.

2.2 Algorithms for Logic Operations

BDDs for the functions of given Boolean expressions. can be generated in the following manner.

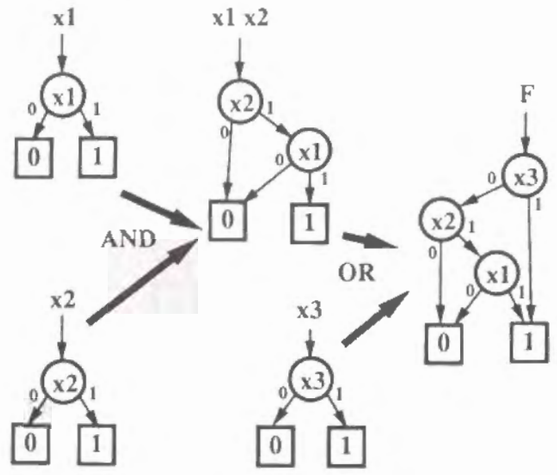


Figure 2.4: Generation of BDDs for $F = (x1 \cdot x2 \vee x3)$.

(address)	(index)	(0-edge)	(1-edge)	
N_0	-	-	-	$\longleftarrow 0$
N_1	-	-	-	$\longleftarrow 1$
N_2	x_1	N_0	N_1	
N_3	x_1	N_1	N_0	$\longleftarrow F_3(=\overline{x_1})$
N_4	x_2	N_0	N_3	$\longleftarrow F_1(=x_2 \cdot \overline{x_1})$
N_5	x_2	N_2	N_3	$\longleftarrow F_2(=x_2 \oplus x_1)$
N_6	x_2	N_3	N_1	$\longleftarrow F_4(=x_2 \vee \overline{x_1})$

Figure 2.5: BDD representation using a table.

1. Define a fixed order of input variables.

2. Make a BDD with a single node for each input variable.

3. Construct more complicated BDDs by applying logic operations on BDDs according to the Boolean expressions.

An example for $F = (x_1 \cdot x_2 \vee x_3)$ is shown in Fig. 2.4. First, trivial BDDs for x_1, x_2, x_3 are generated. Then applying the AND operation between x_1 and x_2, the BDD of $x_1 \cdot x_2$ is generated. The final BDD for the entire expression F is obtained as the result of the OR operation between $x_1 \cdot x_2$ and x_3.

In this section we show the algorithms of logic operations on BDDs.

2.2.1 Data Structure

In a typical implementation of the BDD manipulator, all the nodes are stored in a single table on the main memory of the computer. Each node has three basic attributes; an index of the input variable and two pointers of 0- and 1-edges. Some additional pointers and counters are attached to the node data for maintaining the table. Figure 2.5 shows an example of the table representing BDDs shown in Fig. 2.3. 0- and 1-terminal nodes are at first allocated in the table as the special nodes. By referring the address of a node, we can immediately know whether the node is a terminal or not.

In a shared BDD, isomorphic sub-graphs should be shared without fail. Namely, Two equivalent nodes never coexist. The property is maintained by recording all the nodes in a hash table. Every time we check the hash table before creating a new node. If there already exists a node whose corresponding index, 0-edge, and 1-edge are identical, we do not create a new node but simply copy the pointer to the existing node. This task can be done in a constant time if the hash table acts successfully. The performance of the hash table is important since it is frequently referred in the BDD manipulation.

Using this technique, a Boolean function on a BDD manipulator can be identified by the address of the root node of the BDD. Consequently, we can perform the equivalence checking or tautology checking of Boolean functions by only comparing the addresses of the root nodes of the BDDs, independent of the number of nodes.

2.2.2 Algorithms

Binary Logic Operations

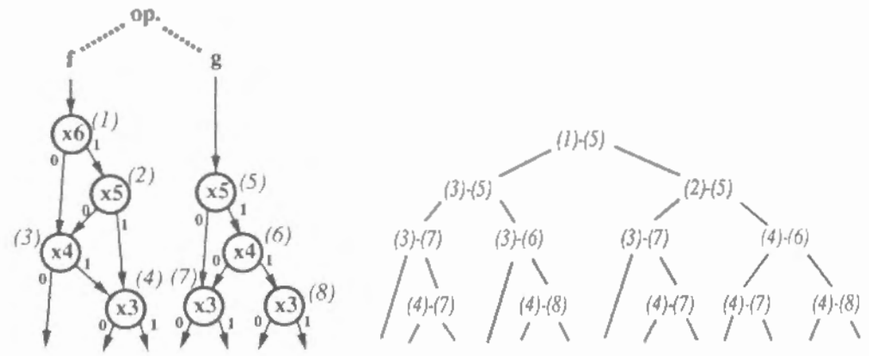
The binary logic operation is the most important part in the techniques of BDD manipulation. Here we show the algorithm of generating a BDD which represents the result of a binary operation $f \circ g$, for given two BDDs of f and g. This algorithm is based on the following formula:

$$f \circ g = \overline{v} \cdot (f_{(v=0)} \circ g_{(v=0)}) \vee v \cdot (f_{(v=1)} \circ g_{(v=1)}),$$

This formula means that the operation can be expanded to two sub-operations $(f_{(v=0)} \circ g_{(v=0)})$ and $(f_{(v=1)} \circ g_{(v=1)})$ with respect to an input variable v. Repeating the expansion recursively for each sub-operation by all the input variables, they are eventually broken down into trivial ones and the results are obtained.

The algorithm of computing h $(= f \circ g)$ is summarized as follows. Here $f.top$ denotes the input variable of the root node of f. f_0 and f_1 are the BDDs pointed by 0-edge and 1-edge from the root node, respectively.

1. When f or g is a constant, or the case of $f = g$:
   return a result according to the kind of the operation.
   (Example) $f \cdot 0 = 0$, $f \vee f = f$, $f \oplus 1 = \overline{f}$



(a) An example. (b) Structure of procedure calls.

Figure 2.6: Procedure of binary operation.

2. If $f.top$ and $g.top$ are identical:
   $h_0 \leftarrow f_0 \circ g_0$; $h_1 \leftarrow f_1 \circ g_1$;
   if $(h_0 = h_1)$ $h \leftarrow h_0$;
   else $h \leftarrow Node(f.top,\ h_0,\ h_1)$;

3. If $f.top$ is higher than $g.top$:
   $h_0 \leftarrow f_0 \circ g$; $h_1 \leftarrow f_1 \circ g$;
   if $(h_0 = h_1)$ $h \leftarrow h_0$;
   else $h \leftarrow Node(f.top,\ h_0,\ h_1)$;

4. If $f.top$ is lower than $g.top$:
   (Compute similarly to 3. by exchanging f and g.)

As mentioned in previous section, we check the hash table before creating a new node to avoid duplication of the node.

Figure 2.6(a) shows an example of a binary operation of BDDs. When we perform the operation between the nodes (1) and (5), the procedure is broken down into the binary tree, as shown in Fig. 2.6(b). Usually we compute this tree in a *depth-first manner*.

It seems that this algorithm always takes an exponential time for the number of inputs since it traverses the binary trees; however, they sometimes contain redundant operations. For example, in Fig. 2.6(b), the operations of (3)-(7), (4)-(7), and (4)-(8) are executed more than once. We can accelerate the procedure using an hash-based cache which memorize the results of recent operations. By referring to the cache before every recursive call, we can avoid duplicate executions for equivalent sub-operations. In this technique, the binary logic operations can be executed in a time almost proportional to the size of BDDs.

The cache size is important to the performance of the BDD manipulation. If it is insufficient, the execution time grows rapidly. Usually we fix the cache size

empirically. In many cases it is several times greater or smaller than the number of the total nodes.

Negation

A BDD for $\overline{f}$, which is the complement of f, has a similar form to the BDD for f, such that just the 0-terminal and the 1-terminal are exchanged. Complemental BDDs contain the same number of nodes, contrasted with the cube set representation, which sometimes suffers a great increase of the data size.

The algorithm of negation on BDDs is described as:

- When f is a constant, return the complement of constant.
- Otherwise, $\overline{f} \leftarrow Node(f.top,\ \overline{f_0},\ \overline{f_1})$.

The computation time is proportional to the numbs of nodes, as well as the binary logic operations.

The computation time for the negation can be improved to a constant time, by using *negative edges*. The negative edges are a kind of attributed edges, discussed in the following section. This technique is now commonly used in many implementation.

Restriction (Cofactoring)

After generating a BDD for f, we sometimes need to compute $f_{(v=0)}$ or $f_{(v=1)}$, such that an input variable is fixed to 0 or 1. This operation is called *restriction*, or *cofactoring*. If v is the highest ordered variable in f, a BDD pointed by 0- or 1-edge of the root node is just returned. Otherwise, we have to expand the BDDs until x become the highest one, and then re-combine them into a BDD. This procedure can be executed efficiently by using the cache technique as the binary operations. The computation time is proportional to the number of nodes which has an index greater than v.

Search for Satisfiable Assignment

After generating BDDs, it is easy to find an assignment for input variables to satisfy the function $f = 1$. If there is a path from the root node to the 1-terminal node, which we call *1-path*, the assignment for variables to activate the 1-path is a solution to $f = 1$. BDDs have am excellent property that every non-terminal node is included in at least one 1-path. (It is obvious since if there is no 1-path, the node should be reduced into the 0-terminal node.) By traversing the BDD from the root node, we can easily find a 1-path in a time proportional to the number of the input variables, independent of the number of the nodes.

In general, there are many solutions to satisfy a function. Under the definition of the costs to assign "1" to respective input variables, we can search an assignment which makes the total cost minimum[LS90]. Namely, where
cost function:

$$Cost = \sum_{i=1}^{n} w_i \cdot x_i \quad (w_i > 0,\ x_i \in \{0,1\}),$$

to seek values for $x_1, x_2, \ldots, x_n$ which makes *Cost* minimum under the constraint $f = 1$. Many NP complete problems can be described in the above format.

Searching for the minimum cost 1-path is implemented based on back tracking of the BDD. It appears to take an exponential time, but we can avoid duplicate tracking for shared subgraphs in the BDD by storing the minimum cost for the subgraph and referring to it at the second visit. This technique eliminates the need to visit each node more than once, so we can find the minimum cost 1-path in a time proportional to the number of nodes in the BDD.

In this method, we can immediately solve the problem if the BDD for the constraint function can be generated in the main memory of the computer. There are many practical examples where the BDD becomes compact. Of course, it is still a problem in NP, so in the worst case the BDD requires an exponential number of nodes and overflows the memory.

We can also efficiently count the number of the solutions to satisfy $f = 1$. On the root node of f, the number of solutions is computed as the sum of the solutions on the two sub-functions f_0 and f_1. By using the cache technique to save the result on each node, we can compute the number of the solutions in a time proportional to the number of nodes in the BDD.

In a similar way, we can compute the truth table density for a given Boolean function represented by a BDD. The truth table density is the rate of 1's in the truth table. This rate indicates the probability to satisfy $f = 1$ for arbitrary assignment to the input variables. Using BDDs, it can be computed as an average of the density for the two sub-functions on each node.

2.2.3 Memory Management

In a typical implementation, the BDD manipulator consumes 20 to 30 Byte of memory for each node. Today there are workstations with more than 100 MByte of memory, and those facilitate us to generate BDDs containing as many as millions of nodes. However, the BDDs still grow large beyond the memory capacity in some practical applications.

In the sequence of logic operations of BDDs, many BDDs for partial results are temporarily generated. It is important for the memory efficiency to delete such already used BDDs. If the BDD to be freed shares sub-graphs with other BDDs, we cannot delete the sub-graphs. In order to determine the necessity of the nodes, we attach a *reference counter* to each node, which shows the number of incoming edges to the node. When a BDD become unnecessary, we decrease the reference counter of the root node, and if it becomes zero, the node can be eliminated. When a node is really deleted, we recursively execute this procedure on its descendant nodes. On the other hand, in the case of copying an edge to a BDD, the reference counter of the root node is increased.

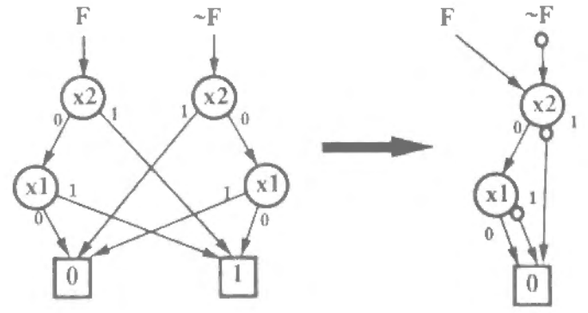


Figure 2.7: Negative edges.

Deletion of the used nodes saves memory space; however, there is a loss of the time because we may eliminate the nodes which will become necessary again later. Besides, deletion of the nodes breaks the consistency of the cache for memorizing recent operation, so we have to recover (or clear) the cache. In order to avoid the loss, although the reference counter becomes zero, we wait to eliminate the nodes until the memory becomes full, and then they are deleted at once as a *garbage collection*.

BDD manipulator is based on the hash table technique, so we have to allocate a fixed size of hash table when initializing the program. In that time, it is difficult to estimate final size of BDDs to be generated, but it is inefficient to allocate too large size of memory space since other application programs cannot use the space. In our implementation, at first a some small size of table is allocated. If the table becomes full during BDD manipulation, the table is re-allocated twice or four times larger, unless the memory overflows. When the table size has reached the limit of extension, the garbage collection process is invoked.

2.3 Attributed Edges

We propose the use of *attributed edges*. Attributed edges are the edges attached with several sorts of attributes each of which is associated with a certain operation. It is regarded as a generalization of the *inverter*[Ake78], or typed edge[MB88]. In this section we introduce three attributes named *negative edge*, *input inverter* and *variable shifter*. We can use these attribute edges in combination.

2.3.1 Negative edges

The *negative edge* is an attribute which indicates to complement the function of the sub-graph pointed by the edge (Fig. 2.7). It is the same idea as Akers's

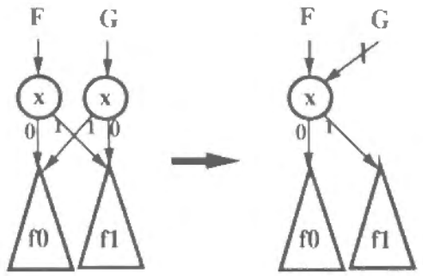


Figure 2.8: Input inverters.

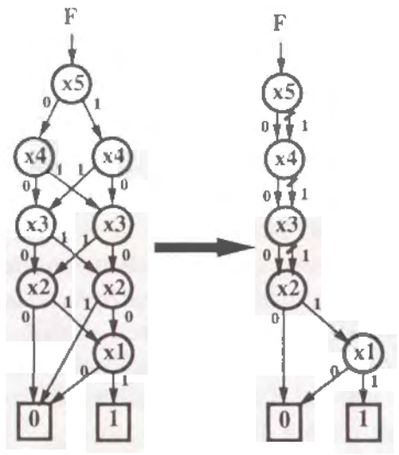


Figure 2.9: An example where input inverters are effective.

inverter[Ake78] and Madre and Billon's *typed edge*[MB88]. The use of negative edges brings the outstanding merits as follows.

- We can reduce the size of BDDs to a half in the best case.
- Negation can be executed without traversing the graph.
- Using the quick negation, we can accelerate logic operations with the rules such as $f \cdot \overline{f} = 0$, $f \vee \overline{f} = 1$, $f \oplus \overline{f} = 1$, etc.
- Using the quick negation, we can transform the operations such as *OR*, *NOR*, *NAND* into *AND* by applying *De Morgan's theorems*, so that we can raise the hit rate of the cache to memorize recent operations.

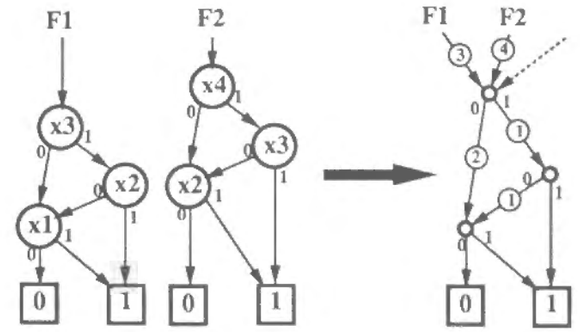


Figure 2.10: Variable shifters.

Abuse of the negative edges breaks the important property that BDDs give canonical representation of Boolean functions. To keep this property, we place the following constraints on the location of the negative edges.

1. Do not use the 1-terminal node. Only use the 0-terminal node.
2. Do not use a negative edges on the 1-edges.

These constraints are basically same as in Madre and Billon's work[MB88].

2.3.2 Input Inverters

We propose another attribute indicating to exchange the 0- and 1-edges at the next node (Fig. 2.8). It is regarded as complementing an input variable of the node, therefore we call it *input inverter*. Using input inverters, we can reduce the size of BDDs to a half in the best case. There are cases where the input inverters are very effective while the negative edges are not so effective (Fig. 2.9).

Since abuse of input inverters also break the property of giving a unique representation, we place a constraint as well as in using negative edges. We use input inverters so that the two nodes f_0 and f_1 pointed by 0- and 1-edges satisfy the constraints: $(f_0 < f_1)$, where '$<$' represents an arbitrary total ordering of all the nodes. In our implementation, each edge identifies the destination with the address of the node table, so we define the order as the value of the address. Under this constraint the uniqueness is maintained in a shared BDD.

2.3.3 Variable Shifters

When there are the two subgraphs which are isomorphic except a difference of their index numbers of input variables, we want to share them into one sub-graph by storing the difference of the indexes. To grant the request, we propose *variable*

shifters, which indicate to add a number to the indexes of its all descendant nodes. In this method, we do not record an information of the index on each node because the variable shifter on each edge have a relative information between a pair of nodes. We place the following rules to use variable shifters.

1. On the edge pointing a terminal node, o not use a variable shifter
2. On the edge pointing the root node of a BDD, the variable shifter indicates the absolute index number of the node.
3. Otherwise, a variable shifter indicates the difference number of the indexes between the start node and the end node of the edge.

For example, the graphs representing $(x_1 \vee (x_2 \cdot x_3))$, $(x_2 \vee (x_3 \cdot x_4)), \ldots, (x_k \vee (x_{k+1} \cdot x_{k+2}))$ can be joined into the same graph as shown in Fig. 2.10.

Using variable shifters, we can reduce the size of BDDs, especially in the case of manipulating a number of regular functions, such as arithmetic systems. The use of variable shifters has another advantage that we can raise the hit rate of the cache of operations, by applying the rule:

$$(f \circ g = h) \Longleftrightarrow (f^{(k)} \circ g^{(k)} = h^{(k)}),$$

where $f^{(k)}$ is a function whose indexes are shifted by k from f, and $\circ$ means a binary logic operation such as *AND, OR, EXOR, etc.*

2.3.4 General Consideration

Here we show that in general the attributed edges keep the property of giving a unique representation of a Boolean function.

The attributed edges can be devised in the following manner. Let S be the set of the Boolean functions of n inputs.

1. Divide S into the two subset S_0 and S_1
2. Define a function $\mathcal{F} : (S \rightarrow S)$, such that for any $f \in S_1$ there is a unique $f_0 \in S_0$ to satisfy $f = \mathcal{F}(f_0)$.

Namely, $\mathcal{F}$ is the operation of the attributed edge, and the partition of S_0 and S_1 is related with the constraint on the location of the attributed edges. We do not use attributed edges pointing to a function in S_0. Using mathematical induction on the number of inputs n, it is obvious that the attributed edge do not break the property of uniqueness.

From the above argument, we can explain both of negative edges and input inverters. Also variable shifters can be explained if we expand the argument as follows.

1. Partition S into a number of subsets $S_0, S_1, \ldots S_n$.
2. For any $k > 1$, Define a function $\mathcal{F}_k : (S \rightarrow S)$, such that for any $f \in S_k$ there is a unique $f_0 \in S_0$ to satisfy $f = \mathcal{F}_k(f_0)$.

Table 2.1: Experimental results

Circuit	Circuit size			#Node	CPU(sec)
	In.	Out.	Nets		
sel8	12	2	29	40	0.3
enc8	9	4	31	33	0.3
add8	18	9	65	49	0.4
add16	33	17	129	97	0.7
mult4	8	8	97	330	0.5
mult8	16	16	418	46594	18.3
c432	36	7	203	89338	34.1
c499	41	32	275	36862	21.5
c880	60	26	464	30548	11.5
c1355	41	32	619	119201	51.4
c1908	33	25	938	39373	22.5
c5315	178	123	2608	40306	29.8

2.4 Implementation and Experiments

In this section, we show the implementation of a BDD package and some experimental results to evaluate the performance of the BDD package and the effect of the attributed edges.

2.4.1 BDD Package

We implemented an BDD program package on Sun3/60 (24Mbyte, SunOS 4.0) using the techniques shown in the previous sections. The program consists of about 800 lines of C codes. This package supports the basic and the essential operations of Boolean functions, as follows.

- Giving the trivial functions, such as 1 (tautology), 0 (inconsistency) and x_k for a given index k.
- Generating BDDs by applying logic operations such as *NOT*, *AND*, *OR*, *EXOR* and *restriction*.
- Equivalence or implication checking between two functions.
- Finding an assignment of inputs to satisfy a function.
- Copying and deleting BDDs.

These operations are modularized as the function of C language. Using them in combination, we can utilize the package for various applications without care of detailed structure of the program.

Table 2.2: Effect of attributed edges

Circuit	(A)		(B)		(C)		(D)	
	#Node	CPU(sec)	#Node	CPU	#Node	CPU	#Node	CPU
sel8	78	0.3	51	0.3	51	0.4	40	0.3
enc8	56	0.3	48	0.3	48	0.3	33	0.3
add8	119	0.4	81	0.4	81	0.4	49	0.4
add16	239	0.7	161	0.6	161	0.7	97	0.6
mult4	524	0.5	417	0.5	400	0.4	330	0.5
mult8	66161	24.8	52750	19.1	50504	19.8	46594	18.3
c432	131299	55.5	104066	36.5	103998	36.8	89338	34.1
c499	69217	22.9	65671	21.3	36986	21.8	36862	21.5
c880	54019	17.5	31378	10.8	30903	11.1	30548	11.5
c1355	212196	89.9	208324	49.3	119465	52.8	119201	51.4
c1908	72537	33.0	60850	21.6	39533	22.3	39373	22.5
c5315	60346	31.3	48353	29.2	41542	28.6	40306	29.8

(A): Using nothing, (B): (A)+ output inverters,
(C): (B)+ input inverters, (D): (C) + variable shifters

In this package we implemented the attributed edges such as negative edges, input inverters and variable shifters. The storage requirement of this package is about 22 bytes a node. We can manage a maximum of about 700,000 nodes in our machine.

2.4.2 Experimental Results

In order to evaluate the efficiency of the program, we made an experiment to generate BDDs from combinational circuits. Notice that in this experiment the BDDs represents the set of the functions of not only primary outputs but all the internal nets. In order to count the number of the nodes exactly, we force to execute garbage collection, which is unnecessary in the practical use.

The results are shown in Table 2.1. The circuit *sel8* is an 8-bit data selecter, and *enc 8* is an 8-bit encoder. The circuits *add8* and *add16* are 8-bit and 16 bit adders, and *mult4*, *mult8* are 4 bit and 8-bit multipliers. The rests are chosen from benchmark circuits in ISCAS '85[BF85].

The sub-column *#Node* shows the number of the nodes in the set of BDDs. *CPU(sec)* shows the total time of loading the circuit data, ordering the input variables, and generating the BDDs.

The results show that we can quickly and compactly represent the functions of these practical circuits. It took less than a minute to represent the circuits of dozens of inputs and hundreds of nets. We can observe that the CPU time is almost proportional to the number of nodes. This manipulator is efficient enough when the size of BDDs is feasible.

In order to evaluate the effect of the attributed edges, we made similar experiments by incrementally applying the techniques, as shown in Table 2.2. The column (A) shows the results of the experiments using original BDDs without any attributed edges. The column (B) shows the results only using negative edges. Comparing the results, we can observe that the negative edges brings a maximum of about 40% reduction in the graph size and outstanding speed up.

The column (C) shows the results in addition to the input inverters to the (B). BDDs are reduced owing to the use of input inverters, and there are no remarkable differences in CPU time. Especially to the circuits *c499*, *c1355* and *c1908*, input inverters are effective and we can observe a maximum about 45% reductions of the graph size, although they are ineffective to some circuits.

The column (D) shows the results in addition to the variable shifters to the (C). BDDs are reduced still more without remarkable differences of the CPU time. Variable shifters are effective especially to the circuits with the regular structures, such as arithmetic logics. We can also observe some degree of effects for other circuits.

The above results show that the combination of the three attributed edges are much effective in many cases, though neither of them is all-round effective alone.

2.5 Remarks and Discussions

In this chapter, we have shown the techniques of BDD manipulation. These techniques have been developed and improved in many laboratories in the world[Bry86, MIY90, MB88, BRB90], and some program packages are opened to public. The techniques of the attributed edges have been tried to improve the efficiency of the programs. Especially, the negative edges are now commonly used because of their remarkable advantage. Using the BDD packages, a number of works are in progress on the VLSI CAD and other various areas in computer science.

The novelty of BDD manipulation are summarized as:

1. Extracting the redundancy which is contained in the Boolean functions by using a fixed variable ordering.

2. Completely removing the redundancy in the two rules:
"no duplicate nodes" and "no equivalent computation again".

The algorithms of BDDs are based on the quick search of the hash tables and the linked list data structure. Both of the two techniques greatly benefit from the property of the *random access machine model*, such that any data on the main memory can be accessed in a constant time. As most of computers are designed in this model, we can conclude that the BDD manipulation algorithms are fairly sophisticated and adapted to the conventional computer model.

Chapter 3

Variable Ordering for BDDs

3.1 Introduction

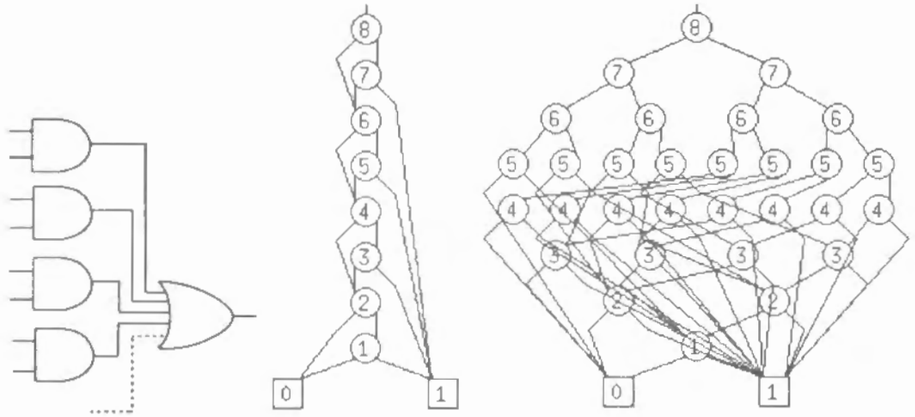
BDDs give canonical forms of Boolean functions provided that the order of input variables is fixed. BDDs can have many different forms for a function by permuting the variables, and sometimes the size of BDDs greatly varies with the order. The size of BDDs decides not only memory requirement but also execution time for their manipulation. The variable ordering algorithm is one of the most important issues in the application of BDDs.

The effect of variable ordering depends on the kind of function to be handled. There are very sensitive examples that the BDD size vary extremely (exponentially to the number of inputs) by only reversing the order. Such functions often appear in practical digital system designs. On the other hand, there are examples that the variable ordering is ineffective. For example, the symmetric functions obviously have the same form for any variable order. It is known that the function of multiplier[Bry91] cannot be represented by a polynomial-sized BDD in any order.

There are some works on the variable ordering. Concerning the method to find the exactly best order, Friedman et al. presented an algorithm[FS87] of $O(n^2 3^n)$ time based on the *dynamic programming*, where n is the number of inputs. It is still difficult to find the best order in a practical time for functions with many inputs, although this algorithm has been improved to the point where the best order can be found for some functions with 17 inputs[ISY91].

From the practical viewpoint, heuristic methods are intensively researched. Malik et al.[MWBSV88] and Fujita et al.[FFK88] showed the heuristic methods based on the topological information of logic circuits. Butler et al. [BRKM91] uses testability measure for the heuristics, which reflect not only topological but logical information of the circuit. These methods are to find a (may be) good order before generating BDDs. They are applied to the practical benchmark circuits and compute a good order in many cases.

Fujita et al.[FMK91] showed another approach that improves the order for the given BDD by repeating the exchange of the variables. It can give further better



(a) Circuit. (b) In the best order. (c) In the worst order.

Figure 3.1: BDDs for 2-level AND-OR circuit.

results than the initial BDDs, but sometimes it is trapped in local optimum.

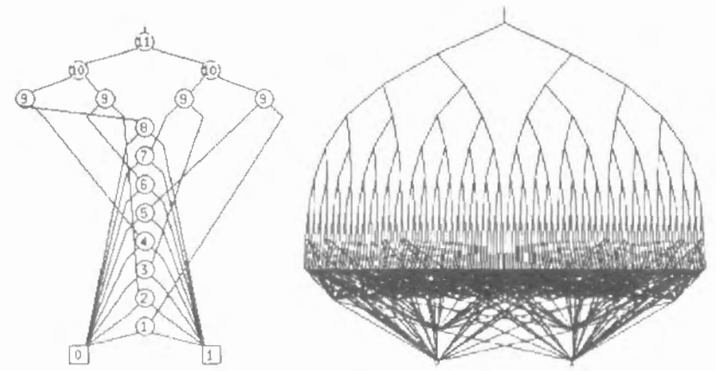
In this chapter, we discuss the properties on the variable ordering for BDDs, and show two heuristic methods of variable ordering which we have developed.

3.2 Properties on the Variable Ordering

Empirically, the following properties are observed on the variable ordering for BDDs.

1. *(Local computability)*
   The group of the inputs with local computability should be near in the order. Namely, we had better keep inputs near that are closely related with each other. For example, we consider the BDD representing the function of the *AND-OR* 2-level logic circuit with $2n$ inputs, as shown in Fig. 3.1(a). It takes $2n$ nodes under the ordering $x_1 \cdot x_2 \vee x_3 \cdot x_4 \vee \ldots \vee x_{2n-1} \cdot x_{2n}$ (Fig. 3.1(b)). On the other hand, it becomes $(2 \cdot 2^n - 2)$ nodes under the order $x_1 \cdot x_{n+1} \vee x_2 \cdot x_{n+2} \vee \ldots \vee x_n \cdot x_{2n}$ (Fig. 3.1(c)).

2. *(Power to control the output)*
   The inputs which greatly affect the function should be located at higher positions (near positions to the root node) in the order. For example, we consider the function of the *8-bit data selector* with three control inputs and eight data inputs. When the control inputs are ordered high, the BDD size is linear (Fig. 3.2(a)), while it becomes an exponential number of nodes using the reversal order (Fig. 3.2(b)).



(a) In the best order. (b) In the worst order.

Figure 3.2: BDDs for 8-bit data selector.

If we could find a variable order which satisfies those two properties, the BDDs would be compact. However, the two properties are mixed ambiguously, and sometimes they require the conflicting orders with each other. In addition, when representing multiple functions together with shared BDDs, there is another problem that those functions may require the different orders. It is difficult to find a point of compromise. For large-scale functions, automatic methods to give an appropriate solution are desired.

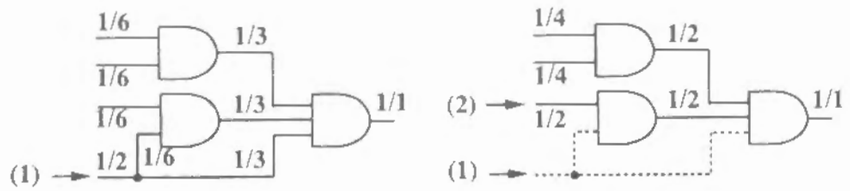
There are two approaches on the heuristic methods of variable ordering:

- To find an appropriate order before generating BDDs by using logic circuit information which is the source of the Boolean function to be represented.

- To reduce BDD size by permuting the variables on a given BDD started from with an initial variable order.

The former approach refers only the circuit information, not using the detailed logical data, to carry out the ordering in a short time. It is one of the most effective ways at present for large-scale problems, although it sometimes gives a poor result depending on the structure of the circuits. On the other hand, the latter approach can find a fairly good order using fully logical information of the BDDs. It is useful when the former method is not available or ineffective. A drawback of this approach is that it cannot start if we fail to make an initial BDD in a reasonable size.

We first propose *Dynamic Weight Assignment (DWA) method*, which belongs the former approach, and show their experimental results. We then present *minimum-width method*, an reordering method that we developed. The two methods can be used in combination.



(a) First assignment. (b) Second assignment.

Figure 3.3: Dynamic weight assignment method.

3.3 Dynamic Weight Assignment Method

When we utilize BDD techniques for digital system design, we first generate BDDs representing the functions for given the logic circuits. If the size of the initial BDD is not feasible, we cannot perform any operation of BDDs. Therefore, it is important for practical use to find a good order before generating BDDs.

Regarding the properties on the variable ordering of BDDs, we have developed *Dynamic Weight Assignment (DWA) method.* In this method, the order is computed from the topological information of a given combinational circuit.

3.3.1 Algorithm

First, we assign a weight 1.0 to one of the primary outputs of the circuit. The weight is then propagated toward primary inputs in the following manner:

1. At each gate, the weight on the output is equally divided and distributed to the inputs.

2. At each fan-out point, the weights of the fan-out branches are accumulated into the fan-out stem.

After this propagation, we give the highest order to the primary input with the largest weight. Since the weights reflects the contribution to the primary output in a topological sense, primary inputs with a large weight are expected to have large influence to the output function.

Next, after choosing the highest input, we delete the part of the circuit which can be reached only from the primary input already chosen, and re-assign the weights from the beginning, to choose the next primary input. By repeating the assignment and deletion of the sub-circuit, we obtain the order of the input variables. An example of this procedure is illustrated in Fig. 3.3(a)(b). When we delete the sub-circuit, the largest weight in the prior assignment is distributed to the neighboring inputs in the re-assignment, and their new weights are increased. Thereby, the neighboring inputs tend to be close to the prior ones in the order.

Table 3.1: Effect of variable ordering

Circuit	(A)		(B)		(C)		(D)	
	#Node	CPU	#Node	CPU	#Node	CPU	#Node	CPU
sel8	40	0.3	41	0.3	390	0.4	57	0.4
enc8	33	0.3	31	0.3	30	0.3	37	0.4
add8	49	0.4	120	0.4	452	0.4	1183	0.6
add16	97	0.7	248	0.5	1700	0.9	94814	24.1
mult4	330	0.5	358	0.4	304	0.5	394	0.5
mult8	46594	18.3	38187	14.5	31026	14.0	77517	26.1
c432	89338	34.1	11348	7.4	6205	5.6	479711	278.6
c499	36862	21.5	68816	39.1	32577	21.0	112815	78.0
c880	30548	11.5	(>500k)		(>500k)		(>500k)	
c1355	119201	51.4	246937	102.9	103301	46.9	373974	179.0
c1908	39373	22.5	47990	22.7	65895	63.3	91082	47.4
c5315	40306	29.8	105200	32.5	(>500k)		(>500k)	

(A): Using dynamic weight assignment method, (B): In the original order.
(C): In the original order (reverse), (D): In a random order

When the circuit has multiple outputs, we have to choose an output to start the weight assignment. We start from the output with the largest depth from the primary inputs. After the ordering, if we have not ordered all the inputs yet, the output with the next largest depth is selected to order the rest of the inputs.

In the above manner, we can obtain a good order in many cases. The time complexity of this method is $O(m \cdot n)$, where m is the number of the gates and n is the number of the primary inputs. Comparing the cost with the contribution to the reduction of BDDs, this complexity is comparatively small.

3.3.2 Experimental Results

We implemented the DWA method using our BDD package on Sun3/60 (24Mbyte, SunOS 4.0), and conducted the experiments to evaluate the effect of the ordering method.

The results are shown in Table 3.1. The circuits are the same ones as used in Section 2.4. The sub-column *#Node* shows the number of the nodes of BDDs. *CPU(sec)* shows the total time of loading the circuit data, ordering the input variables, and generating the BDDs. The column (B),(C) and (D) shows the results of the experiments without the heuristic method of variable ordering. In column (B) we use the original order of the circuit data. In (C) the order is also original but the reverse of (B). We use a random order in (D).

The ordering method is very effective except in a few cases which are insensitive to the order. The random ordering is quite impractical. The original order of the circuit data sometimes gives a good order, but it is a passive way and it can not always brings the good result. We can conclude that our ordering

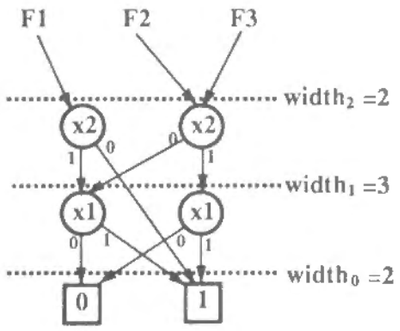


Figure 3.4: Width of BDDs.

method is useful and essential for many practical applications.

3.4 Minimum-Width Method

In this section, we describe another heuristic method of variable ordering based on the reordering after generating BDDs. In the following, n denotes the number of the input variables.

As a reordering method, Fujita et al.[FMK91] presented an incremental algorithm based on the exchange of a pair of variables (x_i, x_{i+1}). Ishiura et al. [ISY91] also showed a *simulated anealing method* with the random exchange of two variables. These incremental search methods have a drawback that they greatly depend on the initial order. If the initial order is far from the best, many exchanges are needed. This takes a long time, and there is the higher risk of being trapped in a bad local minimum solution.

We propose the method with another strategy. At first, we choose one variable based on a certain cost function, and fix it at the highest position (x_n). Next, another variable is chosen from among the rest, and fixed at the second highest position (x_{n-1}). In this manner, all the variables are chosen one by one, and they are fixed from the highest to the lowest. This algorithm has no back tracking. This method is robust to the variation of the initial order. In our method, we define the width of BDDs, as a cost function.

3.4.1 The Width of BDDs

When choosing $x_k (1 \leq k \leq n)$, the variables with higher indexes than k have already been fixed, and the form of the higher part of the graph never varies. Namely, the choice of x_k affects only the part of the graph lower than x_k. The aim on each step is to choose x_k which minimizes the lower part of the graph. It is desired that the cost function should give a good estimation of the min-

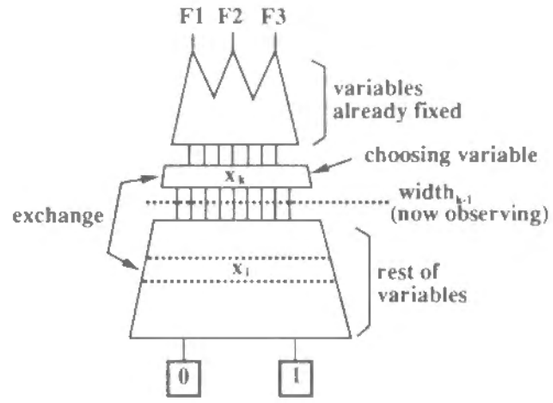


Figure 3.5: Minimum-width method.

imum size of the graph for each choice of x_k. Furthermore, the cost function should be computable within a feasible time. As a cost function to satisfy those requirements, we define the width of BDDs here.

Definition 3.1 *The* width of BDDs at height k, *denoted as* $width_k$, *is the number of the edges crossing the section of the graph between x_k and x_{k+1}, where the edges pointing to the same node are counted as one. The width between x_1 and the bottom of graph is denoted as $width_0$.* □

An example is shown in Fig. 3.4. $width_k$ is sometimes larger than the number of the nodes of x_k since the width also counts the edges which skip the nodes of x_k.

We present the following theorem on the width of BDDs.

Theorem 3.1 *The $width_k$ is constant for any permutation among $\{x_1, x_2, \ldots, x_k\}$ and any permutation among $\{x_{k+1}, x_{k+2}, \ldots, x_n\}$.*

(Proof) $width_k$ represents the total number of sub-functions obtained by assigning all the combinations of Boolean values $\{0,1\}^{n-k}$ into the variables $\{x_{k+1}, x_{k+2}, \ldots, x_n\}$. The total number of sub-functions is independent of the order of the assignment. Therefore, the $width_k$ is constant for any permutation among $\{x_{k+1}, x_{k+2}, \ldots, x_n\}$.

All the sub-functions, obtained above, are uniquely represented in BDDs with a uniform variable order. For any permutation among these variables, the number of the sub-functions does not change because they are still represented uniquely with a different but uniform variable order. Therefore, $width_k$ never varies for any permutation among $\{x_1, x_2, \ldots, x_k\}$. □

3.4.2 Algorithm

Our method uses the width of BDDs to estimate the complexity of the graph, and the variables are chosen one by one from the highest to the lowest by observing the cost function. Namely, we choose x_k which gives minimum $width_{k-1}$ among the rest of the variables, as shown in Fig. 3.5. We call this algorithm *the minimum-width method*. If there are two candidates with the same $width_{k-1}$, we first choose the one at the higher position in the initial order.

It is reasonable to use the width as a cost function because:

- On choosing x_k; $width_{k-1}$ is independent of the order of the lower variables $x_1, x_2, \ldots, x_{k-1}$, as in the above theorem. Therefore, it is robust for the variation of the initial order.

- We should avoid to make $width_{k-1}$ large because $width_{k-1}$ is a lower bound of the number of the nodes at the lower than x_k.

- It is not difficult to compute $width_k$.

The reordering of variables is carried out by repeating the exchange of a pair of variables. The exchange between x_i and x_j can be completed by a sequence of logic operations as:

$$f_{ex} = (\overline{x_i} \cdot \overline{x_j} \cdot f_{00}) \vee (\overline{x_i} \cdot x_j \cdot f_{10}) \vee (x_i \cdot \overline{x_j} \cdot f_{01}) \vee (x_i \cdot x_j \cdot f_{11}),$$

where f_{00}, f_{01}, f_{10}, and f_{11} are the sub-functions obtained by assigning a value 0/1 into the two variables x_i and x_j as:

f_{00}: $x_i = 0$ $x_j = 0$
f_{01}: $x_i = 0$ $x_j = 1$
f_{10}: $x_i = 1$ $x_j = 0$
f_{11}: $x_i = 1$ $x_j = 1$.

This operation requires no traverse on the part of the graph lower than x_i and x_j. The operation time is proportional to the number of the nodes at the higher position than x_i and x_j. Therefore, the higher variables can be exchanged more quickly.

The $width_k$ can be computed by counting the number of sub-functions obtained by assigning any combination of value 0/1 into the variables $x_{k+1}, x_{k+2}, \ldots, x_n$. By traversing the nodes at the higher position than x_k, the $width_k$ can be computed in a time proportional to the number of the visited nodes.

Roughly speaking, the time complexity of our method is $O(n^2G)$, where G is the average size of BDDs during the ordering process. This complexity is considerably less than the conventional algorithms which seeks the exactly best order.

Table 3.2: Experiments of minimum-width method

func.	in.	out.	#node: ave.(min.-max.) init.		after		time (sec)
sel8	12	2	75.2	(19-204)	17.8	(16-20)	0.09
enc8	9	4	25.2	(23-28)	19.9	(19-22)	0.05
adder8	17	9	543.1	(337-938)	40.0	(40-40)	0.53
mult6	6	6	2803.5	(2382-3209)	2145.9	(2123-2296)	6.63
5xpl	7	10	63.9	(58-72)	36.0	(36-36)	0.20
9sym	9	1	23.0	(23-23)	23.0	(23-23)	0.25
alupla	25	5	8101.7	(4178-12952)	1055.0	(856-1178)	33.21
vg2	25	8	861.2	(562-1688)	84.2	(81-87)	1.80

Table 3.3: Experiments for large scale examples

func.	in.	out.	#node init.	after	time (sec)
c432	36	7	23290	1383	177.5
c499	41	32	29702	21962	1311.8
c880	60	26	19100	18336	721.1
c1908	33	25	11083	6590	239.1
c3540	50	22	214941	33975	7493.9
c5315	178	123	27958	2066	14548.3

3.4.3 Experimental Results

We implemented our ordering method as shown above, and conducted some experiments for an evaluation. We used a SPARC Station 2 (SunOS 4.1.1, 32 M Byte). The program is described in C and C++. The memory requirement of BDDs is about 21 Bytes a node.

In our experiments, we generated initial BDDs for given logic circuits in a certain order of the variables and applied our ordering method to the initial BDDs. We used *negative edges*, but there are no serious differences on the performance.

The results for some examples are summarized in Table 3.2. In this table, *sel8*, *enc8*, *add8*, and *mult6* are the same ones as used in Section 2.4. The other items are chosen from the benchmark circuits in DAC'86[dG86]. These circuits have multiple outputs in general. Our program handles multiple output functions using the shared BDD technique. In the re-ordering method, the performance greatly depends on the initial order. We generated 10 initial BDDs in random orders, and applied our ordering method in each case. The table shows the maximum, minimum, and average number of the nodes before and after ordering.

Table 3.4: Comparison with incremental search

func.	init.#node	Min-Width	local search	combination
sel8 (best)	8	12	8	10
(worst)	382	17	38	11
(random)	75.2	17.8	19.7	10.1
adder8 (best)	40	40	40	40
(worst)	1139	40	178	40
(random)	543.1	40.0	182.9	40.0
alupla (best)	830	969	830	830
(worst)	12968	979	2835	830
(random)	8101.7	1055.0	2311.5	998.0

CPU time is the average time of the orderings for 10 cases (including the time of generating initial BDDs).

The results show that our method can reduce the size of BDDs remarkably for the most examples, except for *9sym*, which is a symmetric function. Note that for the various initial order, our method constantly gives good results. This result can be refered in the evaluation of other ordering methods.

Next, the similar experiments were conducted for the larger examples. The functions were chosen from the benchmark circuits in ISCAS'85[BF85]. These circuits are too large and complicated to generate initial BDDs in a random order. We applied DWA method, which is presented in prior section, to obtain a good initial order.

The results are shown in Table 3.3. Our method is also effective for large scale functions in terms of graph reduction, though it takes longer time (but much faster than the methods which seeks the exactly best order). The sizes of the BDDs after reordering are almost equal to the heuristic methods[MWBSV88, FFK88, MIY90, BRKM91] which use circuit information, and our method may be more constantly effective for all the circuits. Remarkably, we find that *c5315* can be represented in only about 2000 nodes, which is far less than the results by any other method (as about 12000 nodes[FMK91]). Our results are useful to evaluate other heuristic methods of variable ordering. The weak points of our method include that it takes longer time than the heuristic methods using the circuit information and that it requires a certain initial BDDs. However, we can say that it is effective to the applications which have many logic operations after generating of BDDs.

We conducted another experiment to compare the properties of the minimum-width method and the incremental search method. We implemented a method which exchanges a pair of variables on the next position if the exchange reduces the size of the BDDs. For the three examples, we applied both ordering methods to the same function for various initial orders including the best and worst ones and the average of trying 10 random orders. The result is shown in Table 3.4.

It shows that the two ordering methods have complementary properties. The incremental search never gives worse results than any initial order, but the effect greatly depends on the initial order. On the other hand, the minimum-width method does not guarantee a better result than the initial orders, but the results are constantly close to the best one.

These properties lead to conclude that it is effective to apply the minimum-width method at first because it seeks a good order with a global view, and then to apply the incremental search for the final optimization. The results of our experiments with such combination, which are summarized in Table 3.4, show that the combination is more effective than applying either of the two methods.

3.5 Conclusion

We have discussed the properties on variable ordering, and shown two heuristic methods: DWA method and minimum-width method. The former one finds an appropriate order before generating BDDs. It refers topological information of the Boolean expression or logic circuit which specifies the sequence of logic operations. Experimental results show that the DWA method finds a tolerable order in a short computation time for many practical circuits. On the other hand, the minimum-width method finds an appropriate order for a given BDD using no additional information. It seeks a good order with a global view, not based on incremental search. In many cases, this method gives better results than the DWA method in a longer but still reasonable computation time.

Based on our discussion of the variable ordering, we can conclude that it is effective to apply the above heuristic methods in the following course.

1. At first, generate an initial BDDs in a order which is given by a topological-based heuristic, such as DWA method.

2. To reduce the size of BDDs, apply an exchange-based heuristic with a global sense, such as the minimum-width method.

3. Try the final optimization by a incremental local search.

This sequence gives fairly good results for many practical problems.

Nevertheless, the above methods are only heuristics, so there are some cases where those methods give poor results. Tani et al.[THY93] has been proved that the algorithm finding the exactly best order needs a time complexity in NP complete. This indicate that it is almost impossible to have an ultimate method of variable ordering to always find best order in a practical time. We will make do with some heuristic methods according to the applications.

The techniques of variable ordering are intensively researched still now. One remarkable work is the *dynamic variable ordering*, recently presented by Rudell[Rud93]. It is based on having the BDD package itself determine and maintain the variable order. Every time when the BDDs grow to a limited size, the re-ordering process is invoked automatically, just like garbage collection. This method is

very effective in terms of the reduction of BDD size although it sometimes takes a long computation time.

Chapter 4

Representation of Multi-Valued Functions

4.1 Representation of Don't Care

In many practical applications related to digital system design, we sometimes deal with not only Boolean values (0, 1) but also *don't care* to mask unnecessary information in the computation. It is an important technique to represent Boolean functions containing *don't care*. In this section, we discuss efficient method for representing *don't care* by using BDDs.

4.1.1 Boolean Functions with Don't Care

A Boolean function with *don't care* is regarded as a function from a Boolean vector input to a ternary-valued output, denoted as:

$$f : \{0,1\}^n \rightarrow \{0,1,d\},$$

where d means *don't care*. Such a function is also called an *incompletely specified Boolean function*. In the following sections, we simply call such a function *ternary-valued function*.

Ternary-valued functions are manipulated by the extended logic operations. The rules of the logic operations between two ternary-valued functions are defined as follows.

NOT

f	$\overline{f}$
0	1
1	0
d	d

AND

$f \cdot g$	0	1	d
0	0	0	0
1	0	1	d
d	0	d	d

OR

$f \vee g$	0	1	d
0	0	1	d
1	1	1	1
d	d	1	d

EXOR

$f \oplus g$	0	1	d
0	0	1	d
1	1	0	d
d	d	d	d

In addition, we define two special unary operations $\lceil f \rceil$ and $\lfloor f \rfloor$ as follows. They are important since these operations are used for abstracting Boolean functions from ternary-valued functions.

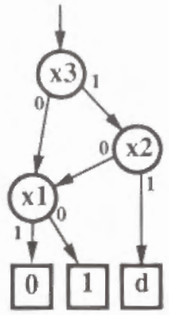


Figure 4.1: A Ternary-valued BDD.

f	$\lceil f \rceil$	$\lfloor f \rfloor$
0	0	0
1	1	1
d	1	0

In the operations of the ternary-valued functions, we sometimes refer to a constant function such that it always returns d. We call it *chaos function*.

4.1.2 Ternary-Valued BDDs and BDD Pairs

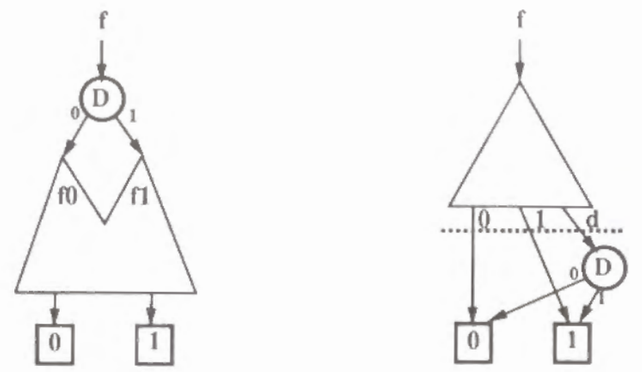
There are two ways to represent ternary-valued functions using BDDs. The first one is to introduce ternary values at the terminal nodes of BDDs, that are '0', '1' and 'd', as shown in Fig. 4.1. We call it *ternary-valued BDD*. This method is natural and easy to understand, However, it has a disadvantages that the operations $\lceil f \rceil$ and $\lfloor f \rfloor$ is not easy, and that we have to develop a new BDD package for ternary-valued functions. Matsunaga et al. reported their work[MF89] which uses such a BDD package.

The second way is to encode the ternary-value into a pair of Boolean values, and represent a ternary-valued function by a pair of Boolean functions, denote as:

$$f : [f_0, f_1].$$

Since each component f_0 and f_1 can be represented by an conventional BDD, we do not have to develop another BDD package. This idea is first presented by Bryant[CB89]. Here we adopted the encoding:

0: $[0,0]$
1: $[1,1]$
d: $[0,1]$,



(a) At the highest position. (b) At the lowest position.

Figure 4.2: D-variable.

which is different one from Bryant's code. The choice of encoding is important for the efficiency of the operations. In this encoding, f_0 and f_1 express the functions $\lfloor f \rfloor$ and $\lceil f \rceil$, respectively. Under this encoding, the constant functions 0 and 1 are expressed as $[0,0]$ and $[1,1]$, respectively. The *chaos* function is represented as $[0,1]$. The logic operations are simply computed as:

$$\overline{[f_0, f_1]} = [\overline{f_1}, \overline{f_0}],$$

$$[f_0, f_1] \cdot [g_0, g_1] = [f_0 \cdot g_0, f_1 \cdot g_1],$$

$$[f_0, f_1] \vee [g_0, g_1] = [f_0 \vee g_0, f_1 \vee g_1].$$

There is a problem which method is more efficient, the ternary-valued BDDs or the BDD pairs. We compare the two methods by introducing *D-variable* in the next section.

4.1.3 D-variable

We propose to use a special variable, which we call *D-variable*, for representing ternary-valued functions. As shown in Fig. 4.2(a), a pair of BDDs $f : [f_0, f_1]$ can be joined into a single BDD using D-variable on the root node whose 0- and 1-edges are pointing f_0 and f_1, respectively. This BDD has elegant properties as follows.

- The constant functions 0 and 1 are represented by the 0- and 1-terminal nodes, respectively. The *chaos* function is represented by a BDD which consists of only one non-terminal node of D-variable.

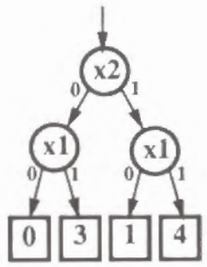


Figure 4.3: A multi-terminal BDD (MTBDD).

- When the function f returns 0 or 1 for any inputs, namely f does not contain *don't care*, f_0 and f_1 become the same function and the sub-graphs are completely shared. In such cases, the D-variable is redundant and automatically removed. Consequently, if f contains no *don't care*, the form becomes the same as a usual BDD.

Using D-variable, we can relate the two representation; the ternary-valued BDDs and the BDD pairs. In Fig 4.2(a), the D-variable is ordered on the highest position in the BDD. When the D-variable is re-ordered to the lowest position, the form of the BDD changes as shown in Fig. 4.2(b). In this BDD, each path from the root node to the 1-terminal node through the D-variable node represents an assignment of input variables to have $f_0 = 0$ and $f_1 = 1$, namely $f = d$ *(don't care)*, and the other paths not through the D-variable node represent the assignments such that $f = 0$ or $f = 1$. (Notice that there are no assignments to have $f_0 = 1$ and $f_1 = 0$.) Therefore, this BDD corresponds to the ternary-valued BDD if we regard the D-variable node as a terminal.

Consequently, we can say that both of the ternary-valued BDDs and the BDD pairs are the special forms of the BDDs using the D-variable, and we can compare the efficiency of the two methods by considering of the properties of variable ordering. From the discussion in previous chapter, we can conclude that the D-variable should be ordered at the higher position (namely use BDD pair) when the D-variable greatly affects the function.

4.2 Representation of Boolean-to-Integer Functions

Extending the argument about ternary-valued functions, we can represent multi-valued logic functions using BDDs. In this section, we deal with the functions from Boolean-vector input to an integer output, denoted as:

$$f : \{0,1\}^n \rightarrow I.$$

B-to-I function

x1	x2	f (f2 f1 f0)
0	0	0 (0 0 0)
0	1	1 (0 0 1)
1	0	3 (0 1 1)
1	1	4 (1 0 0)

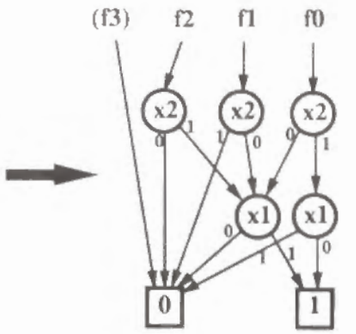
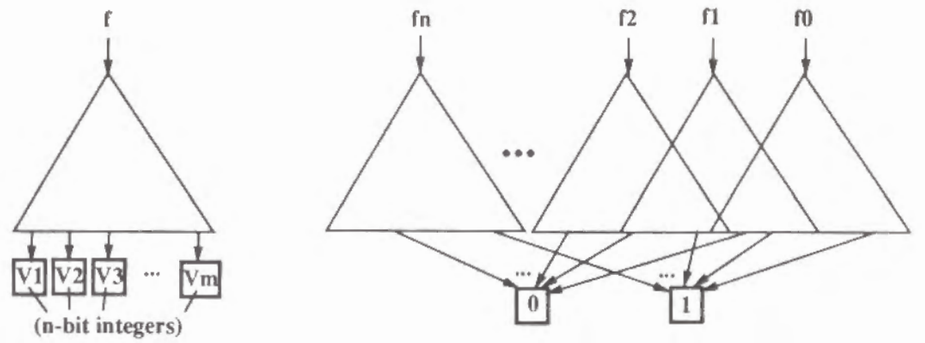


Figure 4.4: A BDD vector.



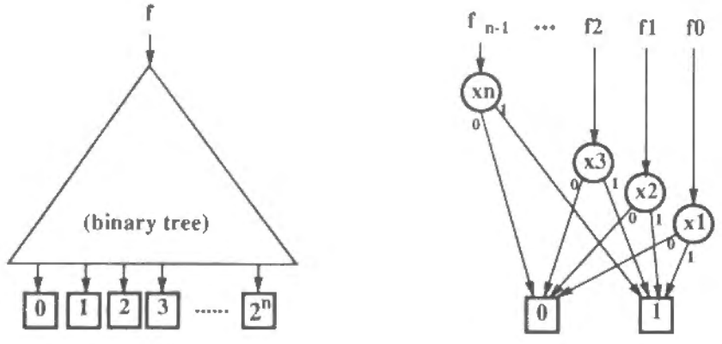
(a) MTBDD. (b) BDD vector.

Figure 4.5: An example where MTBDD is better.

Here we call such functions *Boolean-to-Integer (B-to-I) functions*. Similarly to the ternary-valued functions, there are two ways to represent B-to-I functions using BDDs: *Multi-Terminal BDDs (MTBDDs)* and *BDD vectors*.

MTBDDs are the extended BDDs with multiple terminal nodes, each of which has an integer value (Fig. 4.3). This method is natural and easy to understand; however, we need to develop a new BDD package to manipulate multi-terminals. Hachtel and Somenzi et al. have reported several works[BFG+93, HMPS94] on MTBDDs. They call MTBDD in other words, *Algebraic Decision Diagrams (ADDs)*,

BDD vectors is the way to represent B-to-I functions with a number of usual BDDs. By encoding the integer numbers into n-bit binary codes, a B-to-I function can be decomposed into n pieces of Boolean functions that represent the respective bits as either 1 or 0. These Boolean functions can then be represented



(a) MTBDD. (b) BDD vector.

Figure 4.6: An example where BDD vector is better.

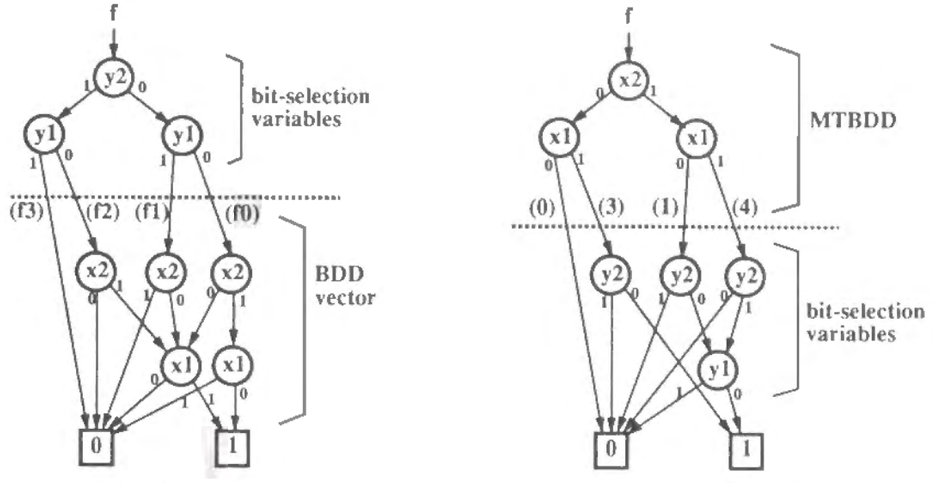
with BDDs which are shared each other (Fig. 4.4). This method was mentioned in [CMZ+93].

Here we discuss which representation is more efficient in terms of size. We show two typical examples that are in contrast to each other.

1. Assume an MTBDD with a large number of non-terminal nodes and a number of terminal nodes with random values of n-bit integers (Fig 4.5(a)). If we represent the same function by using an n-bit BDD vectors, these BDDs can hardly be shared each other since they represent random values (Fig 4.5(b)). In this case, the BDD vector requires about n times of nodes as the MTBDD.

2. Assume a B-to-I function for $(x_1 + 2x_2 + 4x_3 + \ldots + 2^{n-1}x_n)$. This function can be represented with an n-nodes of BDD vector (Fig 4.6(a)). On the other hand, we need 2^n terminal nodes when using MTBDD (Fig 4.6(b)).

Similarly to the ternary-valued functions, we show that the comparison between multi-terminal BDDs and BDD vectors can be reduced to the variable-ordering problem. Assume the BDD shown in Fig. 4.7(a), which was obtained by combining the BDD vector shown in Fig. 4.4 with what we call *bit-selection variables*. If we change the variable order to move the bit-selection variables from higher to lower position, the BDD becomes as shown in Fig. 4.7(b). In this BDD, the sub-graphs with bit-selection variables correspond to the terminal nodes in the MTBDD. Namely, MTBDDs and BDD vectors can be transformed into each other by changing the variable order assuming bit-selection variables. The efficiency of the two representations therefore depends on the nature of the objective functions; we thus cannot determine which representation is more efficient in general.



(a) BDD vector with bit-selection variables. (b) MTBDD with bit-selection variables.

Figure 4.7: Bit-selection variables.

4.3 Remarks and Discussions

In this chapter, we have discussed the methods for representing multi-valued logic functions. We have shown two method of handling *don't care*; ternary-valued BDDs and BDD pairs, and compared the two by introducing the D-variable. The technique of handling *don't care* are basic and important for Boolean function manipulation in many problems. In the method of logic synthesis, which is discussed in the following chapters, the techniques of *don't care* are effectively utilized.

In addition, we extended the argument into B-to-I functions, and presented two methods; MTBDDs and BDD vectors. They can be compared by introducing bit-selection variables, as well as on the ternary-valued functions. Based on the B-to-I function manipulation, we developed arithmetic Boolean expression manipulator, which is presented in Chapter 8.

Recently, several variants of BDDs are devised to represent multi-valued logic functions. The two remarkable works are *Edge-Valued BDDs (EVBDDs)* by Lai et al.[LPV93] and *Binary Moment Diagrams (BMDs)* by Bryant[BC94]. EVBDDs can be regarded as the MTBDDs with attributed edges. EVBDDs contain the attributed edges which indicate to add the value to the functions. Figure 4.8 shows an example for representing a B-to-I function $(x_1 + 2x_2 + 4x_3)$. This technique sometimes effective to reduce memory requirement, especially when representing B-to-I functions for linear expressions.

BMDs provide the representation of algebraic expressions using the similar

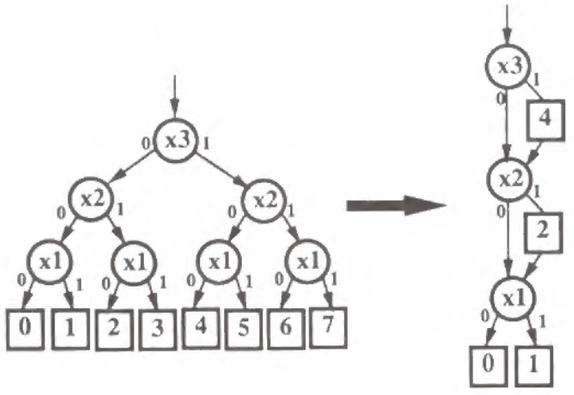


Figure 4.8: EVBDD for $(x_1 + 2x_2 + 4x_3)$.

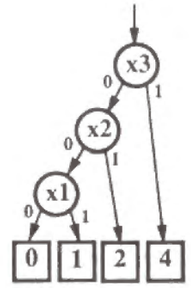


Figure 4.9: BMD for $(x_1 + 2x_2 + 4x_3)$.

structure to MTBDDs. In usual MTBDD, each path from the root node to a terminal node corresponds to an assignment to the input variables, and each terminal node has an output value for the assignment. On the other hand, in the BMD, each path corresponds a product term in the algebraic expression and each terminal node has a coefficient of the product term. For example, a B-to-I function for $(x_1+2x_2+4x_3)$ becomes a binary tree form using MTBDDs; however, the algebraic expression contains only three terms, and it can be represented by a BMD as shown in Fig. 4.9.

Multi-valued logic manipulation is important to broaden the scope of BDD application. Presently, a number of researches are in progress. These techniques are useful not only for VLSI CAD but also for various areas in computer science.

Chapter 5

Generation of Cube Sets from BDDs

5.1 Introduction

In many problems on digital system design, cube sets (also called covers, PLA forms, sum-of-products forms, or two-level logics) are employed to represent Boolean functions. Cube sets have been extensively studied for many years. Cube set manipulation algorithms will assume greater importance as time goes on. In general, it is not so difficult to generate BDDs from cube sets, but there are no efficient methods for generating compact cube sets from BDDs.

In this chapter, we present a fast method for generating prime-irredundant cube sets from BDDs. Prime-irredundant means that each cube is a prime implicant and no cube can be eliminated.

The minimization or optimization of cube sets has received much attention, and a number of efficient algorithms, such as *MINI*[HCO74] and *ESPRESSO*[BHMSV84] have been developed. Since these methods are based on the manipulation of cube sets or truth tables, they cannot be applied to BDD operations directly. Our method is based on the idea of the *recursive operator*, proposed by Morreale[Mor70]. We found that Morreale's algorithm, which is also based on cube set manipulation, can be improved and efficiently adapted for BDD operations.

The features of our method are summarized as follows:

- Prime and irredundant representation can be obtained.
- It generates cube sets from BDDs *directly* without the temporary generation of redundant cube sets in the process.
- It can handle the *don't cares*.
- The algorithm can be extended to manage multiple output functions.

Experimental results show that our method is efficient in terms of time and space. In a practical time, we can generate cube sets consisting of more than

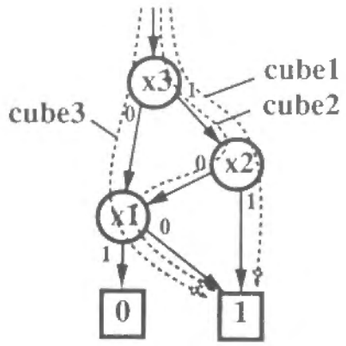


Figure 5.1: 1-path enumeration method.

100,000 cubes and 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method gives the quasi-minimum numbers of cubes and literals, but it does not always give the minimum ones.

In the remainder of this chapter, we first survey a conventional method for generating cube sets from BDDs. Next we present our algorithm to generate prime-irredundant cube sets. We then show experimental results of our method, followed by conclusion.

5.2 Conventional Methods

Akers[Ake78] presented a simple method for generating cube sets from BDDs by enumerating the 1-paths. This method enumerates all the paths from the root node to the 1-terminal node, and lists the cubes which correspond to the assignments of the input variables to activate such paths. In the example shown in Fig. 5.1, we can find the three paths which lead to the cube set:

$$(x_3 \cdot x_2) \vee (x_3 \cdot \overline{x_2} \cdot \overline{x_1}) \vee (\overline{x_3} \cdot \overline{x_1}).$$

In reduced BDDs, all the redundant nodes are eliminated, thereby the literals of the eliminated nodes never appear in the cubes. In the above example, the first cube contains neither x_1 nor $\overline{x_1}$. All of the cubes generated in this method are disjoint because no two paths can be activated simultaneously.

This method can generate disjoint cube sets; however, it does not necessarily give the *minimum* ones. For example, the literal of the root node appears in every cube, but some of them may be unnecessary. In general, considerable redundancy remains in terms of the number of cubes or literals.

Recently, Jacobi and Trullemans[JT92] presented the method of removing such redundancy. The method generates a prime-irredundant cube set from a

BDD in a divide-and-conquer manner. On each node of the BDD, the method generates two cube sets for the two subgraphs of the node, and then combines the two by eliminating redundant literals and cubes. In this method, a cube set is represented with a list of BDDs each of which represents a cube. Each cube is determined whether it is redundant or not by applying BDD operations. This method can generate compact cube sets; however, there is a drawback that it generates lists of redundant cubes temporarily during the procedure, and sometimes large computation time and space are required to manipulate such lists.

5.3 Generation of Prime-Irredundant Covers

In this section, we present the properties of prime-irredundant cube sets, and present the algorithm for generating prime-irredundant cube sets *directly* from given BDDs.

5.3.1 Prime-Irredundant Cube Sets

If a cube set has the following two properties, we call it a prime-irredundant cube set.

- Each cube is a prime implicant; that is, no literal can be eliminated without changing the function.
- There are no redundant cubes. In other words, no cube can be eliminated without changing the function.

For example, the expression $xyz \vee x\overline{y}$ is not prime-irredundant because we can eliminate a literal without changing the function. The expression $xz + \overline{x}\,\overline{y}$ is a prime-irredundant cube set.

Prime-irredundant covers are very compact in general, but they are not always the minimum form. The following three expressions represent the same function and all of them are prime-irredundant.

$$x\overline{y} \vee xz \vee \overline{x}y \vee \overline{x}\,\overline{z}$$
$$x\overline{y} \vee \overline{x}y \vee yz \vee \overline{y}\,\overline{z}$$
$$x\overline{y} \vee \overline{x}\,\overline{z} \vee yz$$

From this observation, we can see that prime-irredundant cube sets do not provide unique forms and that the number of cubes and literals can be different. However, empirically they are not very different from the minimum form in terms of size.

Prime-irredundant cube sets are useful for many applications including logic synthesis, fault testable design, and combinatorial optimization problems.

$ISOP(f(\mathbf{x}))$ {
/* (input) $f(\mathbf{x}) : \{0,1\}^n \rightarrow \{0,1,d\}$ */
/* (output) $isop$: prime-irredundant covers*/
if ($\forall \mathbf{x} \in \{0,1\}^n; f(\mathbf{x}) \neq 1$) { $isop \leftarrow \mathbf{0}$; }
else if ($\forall \mathbf{x} \in \{0,1\}^n; f(\mathbf{x}) \neq 0$) { $isop \leftarrow \mathbf{1}$; }
else {
$v \leftarrow$ one of $\mathbf{x}$;
/* v is the input with highest order in BDD */
$f_0 \leftarrow f(\mathbf{x}\,|_{v=0})$; /* the sub-function on $v = 0$ */
$f_1 \leftarrow f(\mathbf{x}\,|_{v=1})$; /* the sub-function on $v = 1$ */
Compute f_0', f_1' in the following rules;

f_0':

f_1 \ f_0	0	1	d
0	0	1	d
1	0	d	d
d	0	d	d

f_1':

f_1 \ f_0	0	1	d
0	0	0	0
1	1	d	d
d	d	d	d

$isop_0 \leftarrow ISOP(f_0')$;
/* recursively generates cubes including $\overline{v}$ */
$isop_1 \leftarrow ISOP(f_1')$;
/* recursively generates cubes including v */
Let g_0, g_1 be the covers of $isop_0, isop_1$, respectively;
Compute f_0'', f_1'' in the following rules;

f_0'':

g_0 \ f_0	0	1	d
0	0	1	d
1	–	d	d

f_1'':

g_1 \ f_1	0	1	d
0	0	1	d
1	–	d	d

Compute f_d in the following rule;

f_d:

f_1'' \ f_0''	0	1	d
0	0	0	0
1	0	1	1
d	0	1	d

$isop_d \leftarrow ISOP(f_d)$;
/* recursively generates cubes excluding $\overline{v}, v$ */
$isop \leftarrow (\overline{v} \cdot isop_0) \vee (v \cdot isop_1) \vee isop_d$;
}
return $isop$;
}

Figure 5.2: Algorithm for generating prime-irredundant cube sets.

5.3.2 Morreale's Algorithm

Our method is based on the *recursive operator* proposed by Morreale[Mor70]. His algorithm recursively deletes redundant cubes and literals from a given cube set. The basic idea is summarized in this expansion:

$$isop = (\overline{v} \cdot isop_0) \vee (v \cdot isop_1) \vee isop_d,$$

where $isop$ represents a prime-irredundant cube set, and v is one of the input variables. This expansion means that the cube set can be divided into three subsets containing $\overline{v}$, v, and the others. Then, eliminating $\overline{v}$ and v from each cube, the three subsets of $isop_1, isop_0$ and $isop_d$ should also be prime-irredundant. Based on this expansion, the algorithm generates a prime-irredundant cube set recursively (see [Mor70] for details).

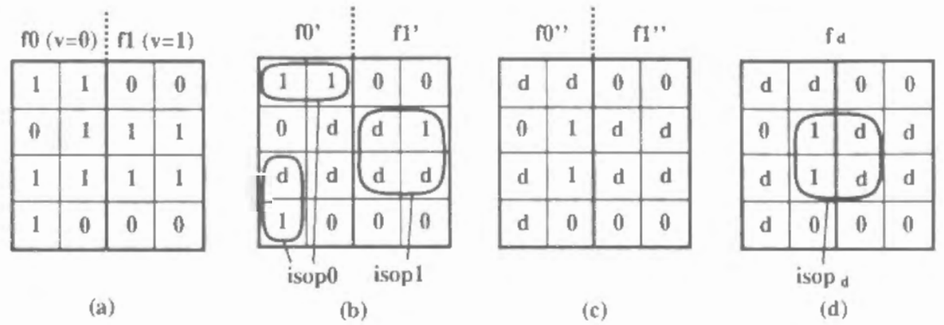


Figure 5.3: An example of using ISOP algorithm.

Unfortunately, Morreale's method is not efficient for large-scale functions because the algorithm is based on cube set representation and it takes a long time to manipulate cube sets for tautology checking, inverting, and other logic operations. However, the basic idea of "recursive expansion" is well suited to BDD manipulation, which is what motivated us to improve and adapt Morreale's method for BDD representation.

5.3.3 ISOP Algorithm Based on BDDs

Our method generates a compact cube set directly from a given BDD, not through redundant cube sets. The algorithm, called *ISOP (Irredundant Sum-Of-Products generation)*, is described in Fig. 5.2. Here we illustrate how it works by using a example shown in Figures 5.3(a) through (d).

In Fig. 5.3(a), the function f is divided into the two sub-functions, f_0 and f_1, by assigning 0, 1 to the input variable ordered at the highest position in the BDD. In (b), f_0' and f_1' are derived from f_0 and f_1 by assigning a *don't care* value to the minterms which commonly give $f_0 = 1$ *and* $f_1 = 1$. f_0' and f_1' represent the minterms to be covered by the cubes including v or $\overline{v}$. We thereby generate their prime-irredundant cube sets $isop_0$ and $isop_1$, recursively. In Fig. 5.3(c), f_0'' and f_1'' are derived from f_0 and f_1 by assigning a *don't care* value to the minterms which are already covered by $isop_0$ or $isop_1$, and in (d) f_d is computed with f_0'' and f_1''. f_d represents the minterms to be covered by the cubes excluding v and $\overline{v}$. We thereby generate its prime-irredundant cube set $isop_d$. Finally, the result of $isop$ can be obtained as the union set of $\overline{v} \cdot isop_0$, $v \cdot isop_1$ and $isop_d$.

Note that in practice the functions are represented and manipulated using BDDs. Here we employ Karnaugh maps to illustrate.

If the order of the input variables is fixed, ISOP algorithm generates a unique form for each function. In other words, it gives a unique form of cube set for a given BDD. Another feature of this algorithm is that it can be applied for functions with don't cares.

This algorithm is well suited for BDD operations because:

- The sub-functions f_0 and f_1 can be derived from f in a constant time.
- Many redundant expansions can be avoided automatically because the redundant nodes are eliminated in reduced BDDs.
- BDDs enables fast tautology checking, which is frequently performed in the procedure.

It is difficult to exactly evaluate the time complexity of this algorithm. In our experiments, as will be shown later, the execution time was almost proportional to the product of the initial BDD size and the final cube set size.

5.3.4 Techniques for Implementation

In our algorithm, ternary-valued functions including the *don't care* value are manipulated. As described in Chapter 4.1, we represent them with a pair of binary functions $(\lfloor f \rfloor, \lceil f \rceil)$. In this method, the tautology checking under a *don't care* condition can be written as $\lceil f \rceil \equiv 1$. The special operation for ternary-valued functions can be computed in the combination of ordinary logic operation for $\lfloor f \rfloor$ and $\lceil f \rceil$. For example, the ternary-valued operation:

f_0':

f_1 \ f_0	0	1	d
0	0	1	d
1	0	d	d
d	0	d	d

can be written as:

$$(\lfloor f_0' \rfloor, \lceil f_0' \rceil) \leftarrow (\lfloor f_0 \rfloor \cdot \overline{\lceil f_1 \rceil},\ \lceil f_0 \rceil).$$

We described earlier that *isop* is obtained as the union set of the three parts, as shown in Fig. 5.2. In order to avoid cube set manipulation, we implemented the method in such a way that the results of cubes are directly dumped out to a file. On each recursive call, we push the processing literal to a stack, which we call a *cube stack*. When a tautology function is detected, the current content of the *cube stack* is appended to the output file as a cube. This approach is efficient because we can only manipulate BDDs, no matter how large the result of the cube set becomes.

Our method can be extended to manage multiple output functions. Sharing the common cubes among different outputs, we obtain more compact representation than if each output were processed separately. In our implementation, the cube sets of all the outputs are generated concurrently; that is, in Fig. 5.2, we extend f to be an array of BDDs to represent a multiple output function. Repeating recursive calls in the same manner as a single output function eventuates in the detection of a multiple output constant which consists of 0's and 1's. The 1's mean that corresponding output functions include the cube which is currently kept in the *cube stack*.

Table 5.1: Comparison with ESPRESSO

Name	In.	Out.	Our method			ESPRESSO		
			Cubes	Literals	Time(s)	Cubes	Lit.	Time(s)
sel8	12	2	17	90	0.3	17	90	0.2
enc8	9	4	17	56	0.2	15	51	0.3
add4	9	5	135	819	0.7	135	819	1.9
add8	17	9	2519	24211	13.3	2519	24211	443.1
mult4	8	8	145	945	1.4	130	874	5.0
mult6	12	12	2284	22274	26.7	1893	19340	1126.2
achil8p	24	1	8	32	0.2	8	32	2.0
achil8n	24	1	6561	59049	8.7	6561	59049	3512.7
5xp1	7	10	72	366	0.8	65	347	1.5
9sym	9	1	148	1036	0.9	87	609	10.7
alupla	25	5	2155	26734	20.5	2144	26632	257.3
bw	5	28	68	374	1.1	22	429	1.4
duke2	22	29	126	1296	3.2	87	1036	28.8
rd53	5	3	35	192	0.3	31	175	0.5
rd73	7	3	147	1024	1.2	127	903	4.2
sao2	10	4	76	575	1.1	58	495	2.4
vg2	25	8	110	914	1.9	110	914	42.8
c432	36	7	84235	969037	1744.8	×	×	>36k
c880	60	26	114299	1986014	1096.6	×	×	>36k

5.4 Experimental Results

We implemented the method described in the foregoing section, and conducted some experiments to evaluate its performance. We used a SPARC Station 2 (SunOS 4.1.1, 32 MByte). The program is written in C and C++.

5.4.1 Comparison with ESPRESSO

First, we generated initial BDDs for the output functions of practical combinational circuits which may be multi-level or multiple output circuits. We then generated prime-irredundant cube sets from the BDDs and counted the numbers of the cubes and literals. We applied the DWA Method, described in Section 3.3, to find the proper order of the input variables for the initial BDD. The computation time includes the time to determine ordering, the formation of initial BDDs, and the time to generate prime-irredundant cube sets.

We compared our results with a conventional cube-based method. We flattened the given circuits into cube sets with the system MIS-II[BSVW87], and then optimized the cube sets by ESPRESSO[BHMSV84].

The results are shown in Table 5.1. For circuits in this work we applied:

Table 5.2: Effect of variable ordering

Name	Heuristic order				Random order			
	#BDD	Cubes	Lit.	Time(s)	#BDD	Cubes	Lit.	Time(s)
sel8	16	17	90	0.3	41	17	90	0.3
enc8	21	17	56	0.2	25	17	56	0.2
add8	41	2519	24211	13.3	383	2519	24211	24.3
mult6	1274	2284	22274	26.7	1897	2354	22963	30.2
achil8n	24	6561	59049	8.7	771	6561	59049	30.9
5xp1	43	72	366	0.8	60	72	364	0.9
alupla	1376	2155	26734	20.4	4309	2155	26730	43.1
bw	85	68	374	1.1	90	64	353	1.1
duke2	396	126	1296	3.2	609	125	1280	3.7
sao2	143	76	575	1.1	133	76	571	1.0
vg2	108	110	914	1.9	1037	110	914	2.7

an 8-bit data selector (sel8), an 8-bit priority encoder (enc8), a 4+4 bit adder (add4), an 8+8 bit adder (add8), a 2×2 bit multiplier (mult4), a 3×3 bit multiplier (mult6), a 24 input *Achilles' heel function*[BHMSV84] (achil8p), and its complement (achil8n). Other items were chosen from benchmarks at MCNC'90.

The table shows that our method is much faster than ESPRESSO, with especially more than 10 times acceleration for the large scale circuits. The speed up was most impressive the c432 and c880, where we generated a prime-irredundant cube set consisting of more than 100,000 cubes and 1,000,000 literals within a reasonable time. We could not apply ESPRESSO to these circuits because we were unable to flatten them into cube sets even after ten hours. In another example, ESPRESSO performed poorly for achil8n because the *Achilles' heel function* requires a great many cubes when we invert it. Here, our method still puts in a good performance because the complementary function can be represented with the same size BDD as the original one.

As far as the number of cubes and literals is concerned, our method in general may give somewhat larger results than ESPRESSO. In most cases, the differences range between 0% and 20%. In none of experiments, did we find an example giving more than two times the difference in terms of number of literals.

5.4.2 Effect of Variable Ordering

We conducted another experiment to evaluate the effect of variable ordering. In general, the size of BDDs depends greatly on the order. We generated prime-irredundant cube sets from the two BDDs of the same function but in a different order: one was in fairly good order (obtained using the *minimum-width method*, shown in Section 3.4), and the other was in a random order.

Table 5.3: Result for variation of the number of inputs (100 random function, single output)

In.	BDD size	Cubes	Literals	Lit./Cubes
1	0.58	0.77	1.35	1.75
2	1.41	1.25	2.84	2.27
3	3.22	2.30	7.17	3.12
4	6.39	4.20	16.05	3.82
5	11.71	7.85	36.39	4.64
6	20.51	14.88	82.18	5.52
7	36.24	27.09	172.06	6.35
8	64.59	52.27	377.41	7.22
9	118.17	99.31	808.09	8.14
10	210.12	192.26	1738.89	9.04
11	365.04	370.90	3693.49	9.96
12	633.97	722.11	7865.91	10.89
13	1144.12	1406.31	16635.79	11.83
14	2154.49	2752.53	35154.84	12.77
15	4151.45	5393.25	73980.57	13.72

As shown in Table 5.2, the numbers of cubes and literals are almost the same for both, while the size of BDDs varies greatly. The result demonstrates that our method is robust for variation in order. However, variable ordering is still important because it affects the execution time and memory requirement.

5.4.3 Statistical Properties

Taking advantage of our method, we examined the statistical properties of prime-irredundant cube sets. We applied our method to 100 patterns of random functions and took the average for the size of initial BDDs and generated cube sets. The random functions were computed using a standard C library.

Table 5.3 shows the property for variation in the number of inputs. Both BDDs and cube sets grow exponentially. It is known that the maximum BDD size is $O(2^n/n)$ (where n is the input number)[Ake78]. Our statistical experiment produced similar results. In terms of number of cubes, we observe about $O(2^n)$. The ratio of cubes to literals (the number of literals per cube) is almost proportional to n.

Table 5.4 shows the property for variation in the number of outputs when the number of inputs is fixed. Both BDDs and cube sets grow a little less proportionally, appearing the effect of sharing their subgraphs or cubes. We expect such data sharing is more effective in practical circuits, since in many cases the output functions are relative each other, unlike random functions. The ratio of cubes to literals is almost constant, as the input number is fixed.

Table 5.4: Result for variation of the number of outputs (inputs = 10)

Out.	BDD size	Cubes	Literals	Lit./Cubes
1	209.80	192.13	1737.84	9.05
2	364.44	381.69	3452.20	9.04
3	500.86	568.10	5145.01	9.06
4	630.93	754.88	6842.25	9.06
5	758.33	933.86	8468.70	9.07
6	884.87	1120.83	10166.36	9.07
7	1011.08	1294.84	11750.90	9.08
8	1136.94	1471.63	13355.59	9.08
9	1262.29	1649.47	14978.33	9.08
10	1388.76	1815.44	16493.02	9.08
11	1513.15	1987.56	18078.64	9.10

Next, we investigated the property for variation in truth-table density, which is the rate of 1's in the truth table. We applied our method to the weighted random functions with 10 inputs, ranging from 0% to 100% in density. Fig. 5.4 shows that the BDD size is symmetric with a center line at 50%, which is like the entropy of information. The number of cubes is not symmetric and peeks at about 60%; however, the number of literals becomes symmetric with BDD size. This result suggests that the number of literals is better as a measure of complexity of Boolean functions than the number of cubes.

5.5 Conclusion

We have presented ISOP algorithm for generating prime-irredundant cube sets directly from given BDDs. The experiments show that our method is much faster than conventional methods. It enables us to generate compact cube sets from large-scale circuits, some of which have never been flattened into cube sets by previous methods. In terms of size of the result, the ISOP algorithm may give somewhat larger results than ESPRESSO, but there are many applications in which such an increase is tolerable. Our method can be utilized to transform BDDs into compact cube sets or to flatten multi-level circuits into two-level circuits.

Cube set representation sometimes requires an extremely large expression which cannot be reduced any more, while the corresponding BDD is quite small. (An n-bit parity function is a good example.) In such cases, our method can generate cube sets as well, but it is hard to use such large-scale cube sets for practical applications. In the following chapters, a compressed representation of cube sets is presented. It allows us to deal with large-scale cube sets efficiently,

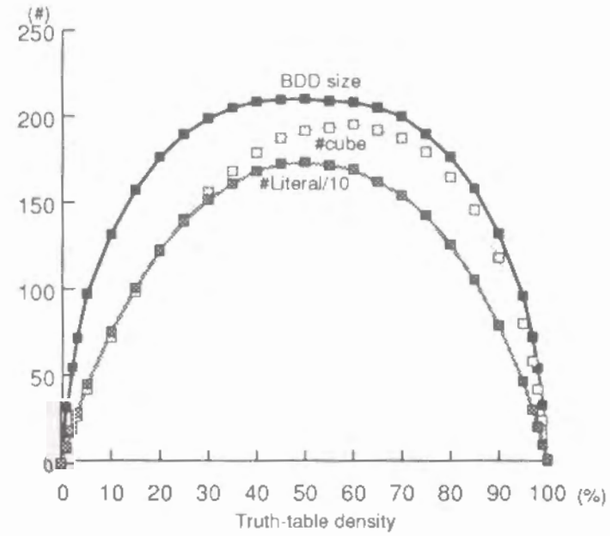


Figure 5.4: Result for variation of truth-table density. (inputs = 10, output = 1)

and the ISOP algorithm can be accelerated still more.

Chapter 6

Zero-Suppressed BDDs

6.1 Introduction

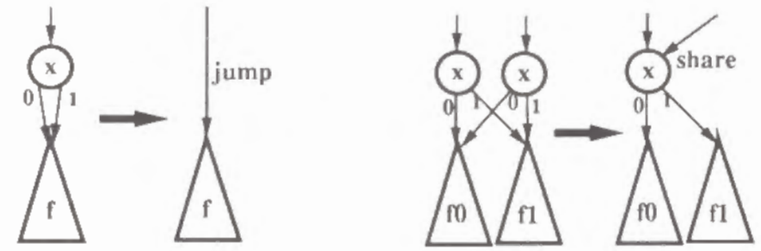
Recently, BDDs have attracted much attention because they enable us to manipulate Boolean functions efficiently in terms of time and space. There are many cases that the algorithm based on conventional data structures can be significantly improved by using BDDs[MF89, BCMD90].

As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* in VLSI CAD problems. By mapping a set of combinations into the Boolean space, they can be represented as a characteristic function using a BDD. This method enables us to implicitly manipulate a huge number of combinations, which have never been practical before. Recently, new two-level logic minimization methods have been developed based on implicit set representation[CMF93]. These techniques are also used to solve general covering problems[LS90]. BDD-based set representation is more efficient than conventional methods. However, it can be inefficient at times because BDDs were originally designed to represent Boolean functions.

In this chapter, we propose a new type of BDD, which has been adapted for set representation[Min93b]. This type of BDD, called a *zero-suppressed BDD (0-sup-BDD)*, enables us to represent sets of combinations more efficiently than using conventional ones. We also discuss *unate cube set algebra*, which is convenient to describe algorithms or procedures of 0-sup-BDDs. We developed efficient methods for computing unate cube set operations, and show some practical applications.

6.2 BDDs for Sets of Combinations

Here we examine the reduction rules of BDDs when applying them to represent sets of combinations. We then show a problem which motivates us to develop a new type of BDDs.



(a) Node elimination. (b) Node sharing.

Figure 6.1: Reduction rules of conventional BDDs.

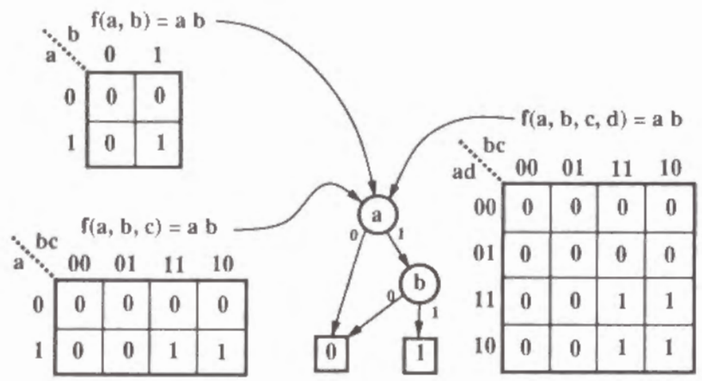


Figure 6.2: Suppression of irrelevant variables in BDDs.

6.2.1 Reduction Rules of BDDs

As mentioned in Chapter 2, BDDs are based on the following reduction rules.

1. Eliminate all the redundant nodes whose two edges point to the same node. (Fig. 6.1(a))

2. Share all the equivalent sub-graphs. (Fig. 6.1(b))

BDDs give canonical forms for Boolean functions under a fixed variable ordering. Most of the works on BDDs are based on the above reduction rules.

It is important how BDDs are shrunk by the reduction rules. One recent paper[LL92] shows that for general (or random) Boolean functions, node sharing makes a much more significant contribution to storage saving than node elimination. However, we consider that the node elimination is also important for practical functions. For example, as in Fig. 6.2, the form of a BDD does not depend on the number of input variables as long as the expressions of the

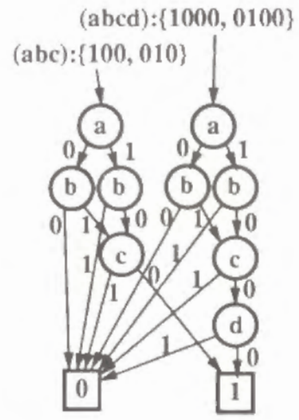


Figure 6.3: BDDs representing sets of combinations.

functions are the same. Using BDDs, the irrelevant variables are suppressed automatically and we do not have to consider them. This property is significant because sometimes we manipulate a function which depends on only a few variables among hundreds. This suppression comes from the node elimination of BDDs.

6.2.2 Sets of Combinations

Presently, there have been many works on BDD applications, but some of them do not use BDDs to simply represent Boolean functions. we are often faced with manipulating *sets of combinations*. Sets of combinations are used for describing solutions to combinatorial problems. We can solve combinatorial problems by manipulating sets of combinations. The representation and manipulation of sets of combination are important techniques for many applications.

A combination among n items can be represented by an n-bit binary vector, $(x_n x_{n-1} \ldots x_2 x_1)$, where each bit, $x_k \in \{1, 0\}$, expresses whether the corresponding item is included in the combination or not. A set of combinations can be represented by a set of the n-bit binary vectors. Sets of combinations can be regarded as subsets of the power set on n items.

We can represent a set of combinations with a Boolean function by using n-input variables for each bit of the vector. The output value of the function expresses whether each combination specified by the input variables are included in the set or not. Such Boolean functions are called *characteristic functions*. The operations of sets, such as union, intersection and difference, can be executed by logic operations on characteristic functions.

Using BDDs for characteristic functions, we can manipulate sets of combination efficiently. In such BDDs, the paths from the root node to the 1-terminal

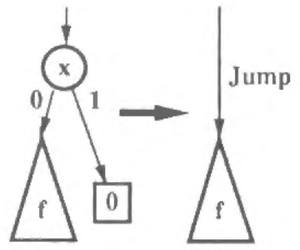


Figure 6.4: New reduction rule for 0-sup-BDDs.

node, which we call *1-paths*, represent possible combinations in the set. Because of the effect of node sharing, BDDs compactly represent sets of combinations with a huge number of elements. In many practical cases, the size of graphs becomes much less than the number of elements. BDDs can be generated and manipulated within a time that is almost proportional to the size of graphs, while previous set representations, such as arrays and sequential lists, require a time proportional to the number of elements.

Set manipulation using BDDs is very efficient, however, there is one inconvenience in that the form of BDDs depends on the number of input variables, as shown in Fig. 6.3. Therefore we have to fix the number of input variables before generating BDDs. This inconvenience comes from the difference in the model on default variables. In sets of combinations, irrelevant objects never appear in any combination, so default variables are regarded as zero when the characteristic function is true. Unfortunately, such variables can not be suppressed in the BDD representation. We have to generate many useless nodes for irrelevant variables when we manipulate very sparse combinations. Node elimination does not work well in reducing the graphs in such cases.

In the following section, we describe a method that solves the above problem by using BDDs based on new reduction rules.

6.3 Zero-Suppressed BDDs

We propose the following reduction rules of BDDs to represent sets of combinations efficiently.

1. Eliminate all the nodes whose 1-edge points to the 0-terminal node. Then connect the edge to the other subgraph directly, as shown in Fig. 6.4.

2. Share all equivalent sub-graphs the same as for original BDDs.

Notice that, contrary to the situation with original BDDs, we do not eliminate the nodes whose two edges point to the same node. This reduction rule is

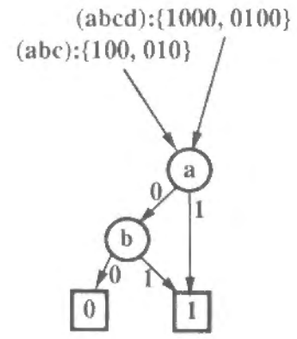


Figure 6.5: 0-sup-BDDs representing sets of combinations.

asymmetric for the two edges, as we do not eliminate the nodes whose 0-edge points to a terminal node.

We call BDDs based on the above rules *Zero-Suppressed BDDs (0-sup-BDDs)*. If the number and the order of input variables are fixed, a 0-sup-BDD uniquely represents a Boolean function. This property is clear because a non-reduced binary tree can be reconstructed from a 0-sup-BDD by applying the reduction rule reversely.

Figure 6.5 shows 0-Sup-BDDs representing the combination sets which are the same ones shown in Fig. 6.3. A feature of 0-Sup-BDDs is that the form is independent of the number of inputs as long as the sets of combinations are the same. We do not have to fix the number of input variables before generating graphs. 0-sup-BDDs automatically suppress the variables which never appear in any combination. It is very efficient when we manipulate very sparse combinations.

For evaluating efficiency of 0-sup-BDDs, we conducted a statistical experiment. We generated a set of one hundred combinations each of which selects k out of 100 objects randomly. We then compared the size of both the 0-sup-BDDs and conventional BDDs representing these random combinations. The result for variation in k is shown in Fig. 6.6. This result shows that 0-sup-BDDs are much more compact than conventional ones especially when k is small. Namely, 0-sup-BDDs are remarkably effective for representing sets of sparse combinations. The effect weakens for large k; however, we can use complement combinations to make k small. For example, the combination selecting 90 out of 100 objects is equivalent to selecting the remaining 10 out of 100.

Another advantage of 0-sup-BDDs is that the number of 1-paths in the graph is exactly equal to the number of combinations in the set. In conventional BDDs, the node elimination breaks this property. Therefore, we conclude that 0-sup-BDDs are more suitable than conventional ones to represent combination sets.

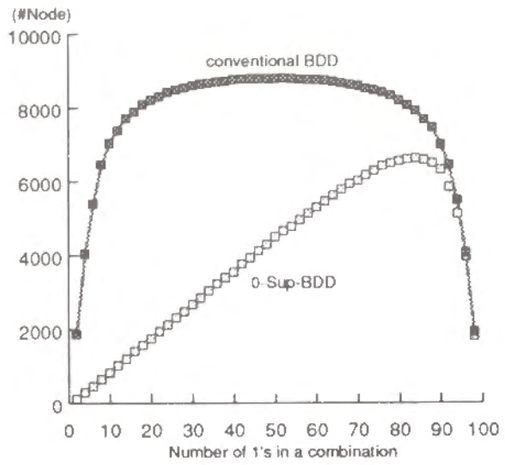


Figure 6.6: Experimental result.

On the other hand, it would be better to use conventional BDDs when representing ordinary Boolean functions, as shown in Fig. 6.2. The difference is in the models on default variables; that is, "fixed to zero" in sets of combinations, and "both the same" in Boolean functions. We can choose one of the two types of BDDs according to the feature of applications.

6.4 Manipulation of 0-Sup-BDDs

In this section, we show that 0-sup-BDDs are manipulated efficiently as well as conventional BDDs.

6.4.1 Basic Operations

In conventional BDDs, we first generate BDDs with only one input variable for each input, and then we construct more complicated BDDs by applying logic operations, such as AND, OR and EXOR. 0-sup-BDDs are also constructed from the trivial graphs by applying basic operations, but the kinds of operations are not the same as with conventional BDDs since 0-sup-BDDs are adapted for sets of combinations.

We arranged the line up of basic operations for 0-sup-BDDs as follows:

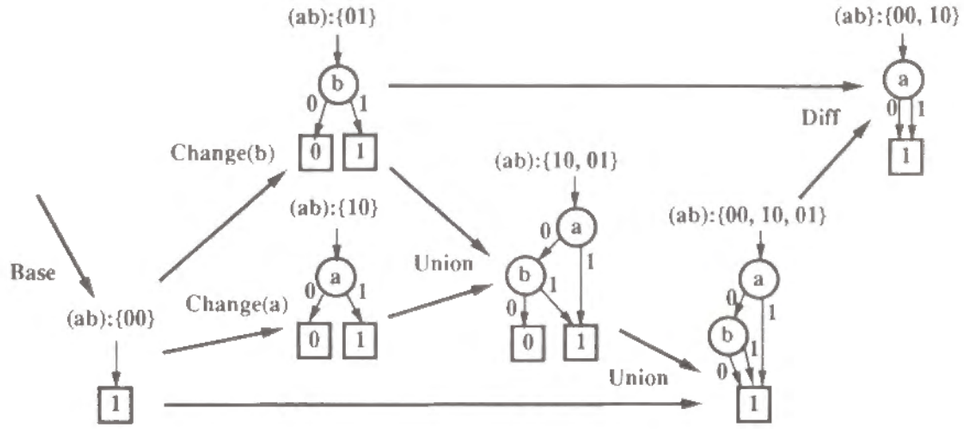


Figure 6.7: Generation of 0-sup-BDDs.

Empty()	returns ϕ. (empty set)
Base()	returns $\{\epsilon\}$.
Subset1(P, var)	returns the subset of P such as $var = 1$.
Subset0(P, var)	returns the subset of P such as $var = 0$.
Change(P, var)	returns P when var is inverted.
Union(P, Q)	returns $(P \cup Q)$
Intsec(P, Q)	returns $(P \cap Q)$
Diff(P, Q)	returns $(P - Q)$
Count(P)	returns $\|P\|$. (number of combinations)

In Fig. 6.7, we show the examples of 0-sup-BDDs generated with the above operations. Empty() returns the 0-terminal node, and Base() is the 1-terminal node. Any one combination can be generated with Base() operation followed by Change() operations for all the variables which appear in the combination. Using Intsec() operation, we can check whether a combination is contained in a set or not.

In 0-sup-BDDs, we do not have NOT operation, which is an essential operation in conventional BDDs. This is reasonable since the complement set $\overline{P}$ cannot be computed if the universal set U is not defined. Using the difference operation, $\overline{P}$ can be computed as $U - P$.

6.4.2 Algorithms

We show here that the above operations can be executed recursively like ones for conventional BDDs.

First, to describe the algorithms simply, we define the procedure Getnode(top, P_0, P_1), which generates (or copies) a node for a variable top and two sub-graphs P_0, P_1.

In the procedure, we use a hash table, called *uniq-table*, to keep each node unique. Node elimination and sharing are managed only by Getnode().

```
Getnode (top, P0, P1) {
    if (P1 == φ) return P0;  /* node elimination */
    P = search a node with (top, P0, P1) in uniq-table;
    if (P exist) return P;   /* node sharing */
    P = generate a node with (top, P0, P1);
    append P to the uniq-table;
    return P;
}
```

Using Getnode(), the operations for 0-Sup-BDDs are described as follows. In the description, *P.top* means a variable with the highest order, and P_0, P_1 are the two subgraphs.

```
Subset1 (P, var) {
  if (P.top < var) return φ;
  if (P.top == var) return P1;
  if (P.top > var)
    return Getnode(P.top, Subset1(P0, var), Subset1(P1, var));
}

Subset0(P, var) {
  if (P.top < var) return P;
  if (P.top == var) return P0;
  if (P.top > var)
    return Getnode(P.top, Subset0(P0, var), Subset0(P1, var));
}

Change (P, var) {
  if (P.top < var) return Getnode(var, φ, P);
  if (P.top == var) return Getnode(var, P1, P0);
  if (P.top > var)
    return Getnode(P.top, Change(P0, var), Change(P1, var));
}

Union (P, Q) {
  if (P == φ) return Q;
  if (Q == φ) return P;
  if (P == Q) return P;
  if (P.top > Q.top) return Getnode(P.top, Union(P0, Q), P1);
  if (P.top < Q.top) return Getnode(Q.top, Union(P, Q0), Q1);
  if (P.top == Q.top)
    return Getnode(P.top, Union(P0, Q0), Union(P1, Q1));
}
```



```
Intsec (P, Q) {
  if (P == φ) return φ;
  if (Q == φ) return φ;
  if (P == Q) return P;
  if (P.top > Q.top) return Intsec(P0, Q);
  if (P.top < Q.top) return Intsec(P, Q0);
  if (P.top == Q.top)
    return Getnode(P.top, Intsec(P0, Q0), Intsec(P1, Q1));
}

Diff (P, Q) {
  if (P == φ) return φ;
  if (Q == φ) return P;
  if (P == Q) return φ;
  if (P.top > Q.top) return Getnode(P.top, Diff(P0, Q), P1);
  if (P.top < Q.top) return Diff(P, Q0);
  if (P.top == Q.top)
    return Getnode(P.top, Diff(P0, Q0), Diff(P1, Q1));
}

Count (P) {
  if (P == φ) return 0;
  if (P == {0}) return 1;
  return Count(P0) + Count(P1);
}
```

These algorithms take an exponential time for the number of variables in the worst case; however, we can accelerate them by using a cache which memorizes results of recent operations in the same manner as it is used in conventional BDDs. By referring the cache before every recursive call, we can avoid duplicate executions for equivalent sub-graphs. With this technique, we can execute these operations in a time that is almost proportional to the size of graphs.

6.4.3 Attributed Edges

In conventional BDDs, we can reduce the execution time and memory requirement by using *attributed edges*[MIY90] to indicate certain logic operations such as inverting. Also 0-sup-BDDs have a kind of attributed edges, but the operation should be different from conventional ones.

Here we present an attributed edge for 0-sup-BDDs. This attributed edge, named *0-element edge*, indicates that the pointing sub-graph has a 1-path which consists of 0-edges only. In other word, a 0-element edge means that the set includes the null-combination "ϵ". We use the notation $\hat{P}$ to express the 0-element edge pointing P.

As with other attributed edges, we have to place a couple of constraints on the location of 0-element edges to keep the uniqueness of the graphs. They are:

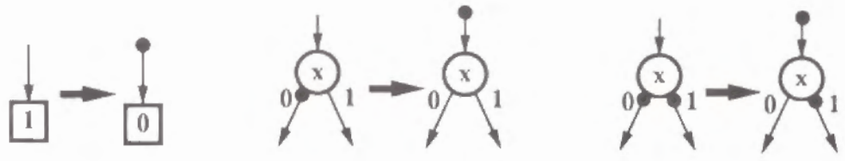


Figure 6.8: Rules of 0-element edges.

- Use the 0-terminal only, since $\{\epsilon\}$ can be written as $\hat{\phi}$
- Do not use 0-element edges at the 0-edge on each node.

If necessary, 0-element edges can be carried over, as shown in Fig. 6.8. The constraint rules can be implemented in the Getnode().

0-element edges accelerate operations on 0-sup-BDDs. For example, the result of Union($P, \{\epsilon\}$) depends only on whether P includes the "ϵ" or not. In such a case, we can get the result in a constant time using 0-element edges, otherwise we have to repeat the expansion until P becomes a terminal node.

6.5 Unate Cube Set Algebra

In this section, we discuss unate cube set algebra for manipulating sets of combinations. A cube set consists of a number of cubes, each of which is a combination of literals. *Unate* cube sets allow us to use only positive literals, not the negative ones. Each cube represents one combination, and each literal represents an item chosen in the combinations.

We sometimes use cube sets to represent Boolean functions; however, they are usually *binate* cube sets containing both positive and negative literals. Binate cube sets have different semantics from unate cube sets. In binate cube sets, literal x and $\overline{x}$ represent $x = 1$ and $x = 0$, respectively, while the absence of a literal means *don't care*, namely $x = 1, 0$, both OK. On the other hand, in unate cube sets, literal x means $x = 1$ and an absence means $x = 0$. For example, the cube set expression $(a + bc)$ represents $(abc) : \{111, 110, 101, 100, 011\}$ under the semantics of binate cube sets, but $(abc) : \{100, 011\}$ under unate cube set semantics.

6.5.1 Basic Operations

Unate cube set expressions consist of trivial sets and algebraic operators. There are three kinds of trivial sets:

0	(empty set),
1	(unit set),
x_k	(single literal set).

The unit set "1" includes only one cube that contains no literals. This set becomes the unit element of the product operation. A single literal set x_k includes only one cube that consists of only one literal. In the following section, a lower-case letter denotes a literal, and an upper-case letter denotes an expression.

We arranged the line-up of the basic operators as follows:

&	(intersection),
+	(union),
−	(difference),
*	(product),
/	(quotient of division),
%	(remainder of division).

(We may use a comma "," instead of "+", and sometimes omit "*".) The operation "*" generates all possible concatenations of two cubes in respective cube sets. Examples of calculation are shown below.

$$\begin{aligned}
\{ab, b, c\} \& \{ab, 1\} &= \{ab\} \\
\{ab, b, c\} + \{ab, 1\} &= \{ab, b, c, 1\} \\
\{ab, b, c\} - \{ab, 1\} &= \{b, c\} \\
\{ab, b, c\} * \{ab, 1\} &= (ab * ab) + (ab * 1) + (b * ab) \\
&\quad + (b * 1) + (c * ab) + (c * 1) \\
&= \{ab, abc, b, c\}
\end{aligned}$$

There are the following formulas in the unate cube calculation.

$$\begin{aligned}
&P + P = P \\
&a * a = a, \quad (P * P \neq P \quad \text{in general}) \\
&(P - Q) = (Q - P) \Longleftrightarrow (P = Q) \\
&P * (Q + R) = (P * Q) + (P * R)
\end{aligned}$$

Dividing P by Q acts to seek out the two cube sets P/Q (quotient) and $P\%Q$ (remainder) under the equality $P = Q*(P/Q)+(P\%Q)$. In general this solution is not unique. Here, we apply the following rules to fix the solution with reference to the *weak-division method*[BSVW87].

1. When Q includes only one cube, (P/Q) is obtained by extracting a subset of P, which consists of cubes including Q's cube, and then eliminating Q's literals from the subset. For example,

   $$\{abc, bc, ac\}/\{bc\} = \{a, 1\}.$$

2. When Q consists of multiple cubes, (P/Q) is the intersection of all the quotients dividing P by respective cubes in Q. For example,

   $$\{abd, abe, abg, cd, ce, ch\}/\{ab, c\}$$

$$= (\{abd, abe, abg, cd, ce, ch\}/\{ab\}) \ \& \ (\{abd, abe, abg, cd, ce, ch\}/\{c\})$$
$$= \{d, e, g\} \& \{d, e, h\}$$
$$= \{d, e\}.$$

3. $(P\%Q)$ can be obtained by calculating $P - P * (P/Q)$.

These three trivial sets and six basic operators are used to represent and manipulate sets of combinations. In Section 6.4, we defined other three basic operations of Subset1(), Subset0() and Change() for assigning a value to a literal; however, we do not have to use the three operations since the weak-division operation can be used as *generalized cofactor* for 0-sup-BDDs. For example, Subset1(P, x_k) can be described as $(P/x_k) * x_k$, and Subset0(P, x_k) becomes $(P\%x_k)$. Change() operation can also be described using some multiplication and division operators. Using unate cube set expressions, we can elegantly express the algorithms or procedures for manipulating sets of combinations.

6.5.2 Algorithms

We show here that the above operations can be efficiently executed using 0-sup-BDD techniques. The three trivial cube sets are represented by simple 0-sup-BDDs. The empty set "0" becomes the 0-terminal, and the unit set "1" is the 1-terminal node. A single literal set x_k corresponds to the single-node graph pointing directly to the 0 and 1-terminal node. The intersection, union, and difference operations are the same as the basic operations of the 0-sup-BDDs shown in Section 6.4. The other three operations, product, quotient, and remainder, are not included in the basic ones. We have developed the algorithms for computing these operations.

If we calculate the multiplication and division operations by processing each cube one by one, the computation time will depends on the length of expressions. Such a procedure is impractical when we deal with very large number of cubes. We developed new recursive algorithms based on 0-sup-BDDs to efficiently calculate large size of expressions.

Our algorithms are based on the divide-and-conquer method. Suppose x is the highest-ordered literal, P and Q are then factored into two-part:

$$P = x * P_1 + P_0, \quad Q = x * Q_1 + Q_0.$$

The product $(P * Q)$ can be written as:

$$(P * Q) = x * (P_1 * Q_1 + P_1 * Q_0 + P_0 * Q_1) + P_0 * Q_0.$$

Each sub-product term can be computed recursively. The expressions are eventually broken down into trivial ones and the results are obtained. In the worst case, this algorithm would require an exponential number of recursive calls for the number of literals; however, we can accelerate them by using a hash-based cache which memorizes results of recent operations. By referring


```
procedure(P * Q)
{   if (P.top < Q.top) return (Q * P) ;
    if (Q = 0) return 0 ;
    if (Q = 1) return P ;
    R ← cache("P * Q") ; if (R exists) return R ;
    x ← P.top ; /* the highest variable in P */
    (P_0, P_1) ← factors of P by x ;
    (Q_0, Q_1) ← factors of Q by x ;
    R ← x (P_1 * Q_1 + P_1 * Q_0 + P_0 * Q_1) + P_0 * Q_0 ;
    cache("P * Q") ← R ;
    return R ;
}
```

Figure 6.9: Algorithm for product.

```
procedure(P/Q)
{   if (Q = 1) return P ;
    if (P = 0 or P = 1) return 0 ;
    if (P = Q) return 1 ;
    R ← cache("P/Q") ; if (R exists) return R ;
    x ← Q.top ; /* the highest variable in Q */
    (P_0, P_1) ← factors of P by x ;
    (Q_0, Q_1) ← factors of Q by x ; /* (Q_1 ≠ 0) */
    R ← P_1/Q_1 ;
    if (R ≠ 0) if (Q_0 ≠ 0) R ← R & P_0/Q_0 ;
    cache("P/Q") ← R ;
    return R ;
}
```

Figure 6.10: Algorithm for division.

to the cache before every recursive call, we can avoid duplicate executions for equivalent subsets. Consequently, the execution time depends on the size of 0-sup-BDDs, not on the number of cubes and literals. This algorithm is shown in detail in Fig. 6.9.

The quotient of division is computed in the same recursive manner. Suppose x is a literal at the root-node in Q, and P_0, P_1, Q_0, Q_1 are the sub cube sets factored by x. (Notice that $Q_1 \neq 0$ since x appears in Q.) The quotient (P/Q) can be described as:

$$(P/Q) = (P_1/Q_1), \quad \text{when} \quad Q_0 = 0.$$
$$(P/Q) = (P_1/Q_1) \ \& \ (P_0/Q_0), \quad \text{otherwise.}$$

Each sub-quotient term can be computed recursively. Whenever we find that one of the sub-quotients $(P_1/Q_1) or (P_0/Q_0)$ results in 0, $(P/Q) = 0$ becomes obvious and we no longer need to compute it. Using the cache technique avoids duplicate

```
***** Unate Cube set Calculator (Ver. 1.1) *****
ucc> symbol a(2) b(1) c(2) d(3) e(2)
ucc> F = (a + b) (c + d + e)
ucc> print F
 a c, a d, a e, b c, b d, b e
ucc> print .factor F
 ( a + b ) ( c + d + e )
ucc> print .matrix F
  1.1..
  1..1.
  1...1
  .11..
  .1.1.
  .1..1
ucc> print .count F
   6
ucc> print .size F
   5 (10)
ucc> G = F * a + c d e
ucc> print G
 a b c, a b d, a b e, a c, a d, a e, c d e
ucc> print .factor G
 a ( b + 1 ) ( c + d + e ) + c d e
ucc> print F & G
 a c, a d, a e
ucc> print F - G
 b c, b d, b e
ucc> print G - F
 a b c, a b d, a b e, c d e
ucc> print  G / (a b)
 c, d, e
ucc> print  G % (a b)
 a c, a d, a e, c d e
ucc> print .mincost G
 a c (4)
ucc> exit
```

Figure 6.11: Execution of unate cube set calculator.

executions for equivalent subsets. This algorithm is illustrated in Fig. 6.10. The remainder $(P\%Q)$ can be determined by calculating $P - P * (P/Q)$.

6.6 Implementation and Applications

Based on the techniques mentioned above, we developed a *Unate Cube set Calculator (UCC)*. This program is an interpreter with a lexical and syntax parser for calculating unate cube set expressions using 0-sup-BDDs. Our program allows up to 65,535 different literals. An example of execution is shown in Fig. 6.11.

We can define the cost for each literal, for use in computing the minimum-cost cube. After constructing 0-sup-BDDs, the minimum-cost cube can be found in a time proportional to the number of nodes in the graph, as using conventional BDDs[LS90].

Because the unate cube set calculator can generate huge 0-sup-BDDs with

Figure 6.12: Results on N-queens problems.

N	Lit.	Sol.	BDD	ZBDD	(B/Z)	(Z/S)
4	16	2	29	8	3.6	4.0
5	25	10	166	40	4.2	4.0
6	36	4	129	24	5.4	6.0
7	49	40	1098	186	5.9	4.65
8	64	92	2450	373	6.6	4.05
9	81	352	9556	1309	7.3	3.72
10	100	724	25944	3120	8.3	4.31
11	121	2680	94821	10503	9.0	3.92
12	144	14200	435169	45833	9.5	3.23
13	169	73712	2044393	204781	10.0	2.78

(B/Z) BDD/ZBDD, (Z/S) ZBDD/Solution.

millions of nodes, limited only by memory capacity, we can manipulate large-scale and complicated expressions. Here we show several applications for the unate cube set calculator.

6.6.1 8-Queens Problem

The 8-queens problem is an example in which using unate cube set calculation is more efficient than using ordinary Boolean expressions.

First, we allocate 64 logic variables to represent the squares on a chessboard. Each variable denotes whether or not there is a queen on that square. The problem can be described with the variables as follows:

- Only one variable is "1" in a particular column.
- Only one variable is "1" in a particular row.
- One or no variable is "1" on a particular diagonal line.

By unate cube set calculation, we can efficiently solve the 8-queens problem. The algorithm can be written as:

$$
\begin{aligned}
S_1 &= x_{11} + x_{12} + \ldots + x_{18} \\
S_2 &= x_{21}(S_1 \% x_{11} \% x_{12}) + x_{22}(S_1 \% x_{11} \% x_{12} \% x_{13}) \\
&\quad + \ldots + x_{28}(S_1 \% x_{17} \% x_{18}) \\
S_3 &= x_{31}(S_2 \% x_{11} \% x_{13} \% x_{21} \% x_{22}) \\
&\quad + x_{32}(S_2 \% x_{12} \% x_{14} \% x_{21} \% x_{22} \% x_{23}) \\
&\quad + \ldots + x_{38}(S_2 \% x_{16} \% x_{18} \% x_{27} \% x_{28}) \\
S_4 &= \ldots
\end{aligned}
$$

These expressions means the strategy as:

S_1: Search all the choices to put the first queen.

S_2: Search all the choices to put the second queen, considering the first queen's location.

S_3: Search all the choices to put the third queen, considering the first and second queen's location.

...

S_8: Search all the choices to put the eighth queen, considering the other queens' locations.

Calculating these expressions with 0-sup-BDDs provides the set of solutions to the 8-queens problem. H. Okuno[Oku94] reported experimental results for N-queens problems to compare 0-sup-BDDs and conventional BDDs. In Table 6.12, the column "BDD" shows the size of BDDs using Boolean algebra, and "ZBDD" is the size of 0-sup-BDDs using unate cube set algebra. This shows that there are about N times less 0-sup-BDDs than conventional BDDs. We can represent all the solutions at once within a storage space almost proportional to the number of solutions.

6.6.2 Fault Simulation

N. Takahashi et al. proposed a method of fault simulation for multiple faults using BDDs[TIY91]. This is a deductive method for multiple faults, that manipulates sets of multiple stuck-at faults using BDDs. It propagates the fault sets from primary inputs to primary outputs, and eventually obtains the detectable faults at primary outputs. The study [TIY91] used conventional BDDs, however; we can more simply compute the fault simulation using 0-sup-BDDs based on unate cube set algebra.

First, we generate the whole set of multiple faults that is assumed in the simulation. The set of all the single stuck-at faults is expressed as:

$$F_1 = (a_0 + a_1 + b_0 + b_1 + c_0 + c_1 + ...),$$

where a_0 and a_1 represent the stuck-at-0 and -1 faults, respectively, at the net a. Other literals are similar. We can represent the set of double and single faults F_2 as $(F_1 * F_1)$. Further, $(F_2 * F_1)$ gives the set of three or less multiple faults. If we assume exactly double (not including single) faults, we can calculate $(F_2 - F_1)$. In this way, the whole set U can easily be described with unate cube set expressions.

After computing the whole set U, we then propagate the detectable fault set from the primary inputs to the primary outputs. As illustrated in Fig. 6.13(a), two faults x_0 and x_1 are assumed at a net x. Let X and X' be the detectable fault sets at the source and sink, respectively, of the net x. We can calculate X' from X with the following unate cube expression as:

$$X' = (X + (U/x_1) * x_1)\%x_0, \text{ when } x = 0 \text{ in a good circuit.}$$
$$X' = (X + (U/x_0) * x_0)\%x_1, \text{ when } x = 1 \text{ in a good circuit.}$$

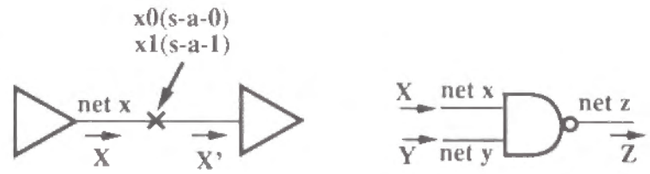


Figure 6.13: Propagation of fault sets

On each gate, we calculate the fault set at the output of the gate from the fault sets at the inputs of the gate. Let us consider a NAND gate with two inputs x and y, and one output z, as shown in Fig. 6.13(b). Let X, Y and Z be the fault sets at x, y and z. We can calculate Z from X and Y by the simple unate cube set expressions as follows:

$$Z = X \,\&Y, \text{ when } x = 0, y = 0, z = 1 \text{ in a good circuit.}$$
$$Z = X - Y, \text{ when } x = 0, y = 1, z = 1 \text{ in a good circuit.}$$
$$Z = X + Y, \text{ when } x = 1, y = 1, z = 0 \text{ in a good circuit.}$$

We can compute the detectable fault sets by calculating those expressions for all the gates in the circuit. Using unate cube set algebra, we can simply describe the fault simulation procedure and can directly execute it by a unate cube set calculator.

6.7 Conclusion

We have proposed 0-Sup-BDDs, which are BDDs based on a new reduction rule, and presented their manipulation algorithms and applications. 0-Sup-BDDs can represent sets of combinations uniquely and more compactly than conventional BDDs. The effect of 0-Sup-BDDs is remarkable especially when manipulating sparse combinations. Based on the 0-sup-BDD techniques, we have discussed the method for calculating unate cube set algebra. We have developed a unate cube set calculator, which can be applied to many practical problems.

Unate cube sets have different semantics from binate cube sets; however, there is a way to simulate binate cube sets using unate ones. We use two unate literals x_1 and x_0 for one binate literal. For example, a binate cube set $(a\ \overline{b} + c)$ is expressed as the unate cube set $(a_1 b_0 + c_1)$. In this way, We can easily simulate the cube-based algorithms implemented in the logic design systems such as ESPRESSO and MIS[BSVW87]. Utilizing this technique, we have developed a practical multi-level logic optimizer. It is detailed in the next chapter.

Unate cube set expressions are suitable for representing sets of combinations, and they can be efficiently manipulated using 0-sup-BDDs. For solving

some types of combinatorial problems, our method are more useful than using conventional BDDs. We expect the unate cube set calculator to be utilized as a helpful tool in researching and developing VLSI CAD systems and other various applications.

Chapter 7

Multi-Level Logic Synthesis Using 0-Sup-BDDs

7.1 Introduction

Logic synthesis and optimization techniques have been used successfully for practical design of VLSI circuits in recent years. Multi-level logic optimization is important in logic synthesis systems and a lot of research in this field has been undertaken[MKLC87, MF89, Ish92]. In particular, *algebraic logic minimization methods*, such as MIS[BSVW87], is the most successful and prevalent way to attain this optimization. This method is based on cube set (or two-level logic) minimization and generates multi-level logic from cube sets by applying a *weak-division* method. This approach is efficient for functions that can be expressed in a feasible size of cube set. Unfortunately, we are sometimes faced with functions whose cube set representations grow exponentially large in respect to the number of inputs. Parity functions and full-adders are such examples. This is a problem of the cube-based logic synthesis methods.

The use of BDDs provided a break-through for that problem. By mapping a cube set into the Boolean space, a cube set can be represented as a Boolean function using a BDD. With this method, we can represent a huge number of cubes implicitly in a small storage space. This enables us to manipulate very large cube sets which have never been practicable before. Based on the BDD-based cube set representation, new cube set minimization methods have been developed [CMF93, MSB93].

BDD-based cube representation is more efficient than other methods. However, it can be inefficient at times because BDDs were originally designed to represent Boolean functions. We have recently developed 0-sup-BDDs which are adapted for representing sets of combinations, as described in the previous chapter. 0-sup-BDD enables us to represent cube sets more efficiently. They are especially effective when we manipulate cube sets using intermediate variables to represent multi-level logic.

In this chapter, we presents a fast weak-division algorithm for implicit cube

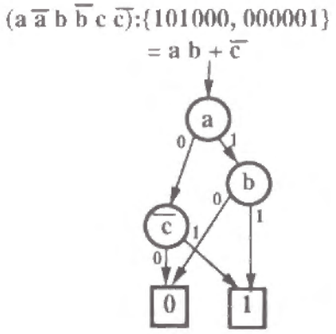


Figure 7.1: Implicit cube set representation based on 0-sup-BDDs.

sets based on 0-sup-BDDs. This algorithm can be computed in a time almost proportional to the size of the 0-sup-BDDs, which are usually much smaller than the number of literals in the cube set. By using this method, we can quickly generate multi-level logic from cube sets even for parity functions and full-adders, which have never been possible with the conventional algebraic methods. We implemented a new multi-level logic synthesizer using the implicit weak-division method. Experimental results indicate our method is much faster than conventional methods and differences are more significant when larger cube sets are manipulated. The implicit weak-division method is expected to accelerate logic synthesis systems significantly and enlarge the scale of applicable circuits.

The following sections, we first discusses the implicit cube set representation based on 0-sup-BDDs. We then present the implicit weak-division method and show experimental results.

7.2 Implicit Cube Set Representation

Cube sets (also called covers, PLAs, sum-of-products forms and two-level logic) are employed to represent Boolean functions in many digital system design, testing, and problems in artificial intelligence. In a cube sets, each cube is formed by a combination of positive and negative literals for input variables. To be exact, it is a *binate* cube set, different from a *unate* cube set discussed in Chapter 6. In this section, we present an implicit method for representing cube sets using 0-sup-BDDs and a method for generating prime-irredundant covers using the implicit representation.

7.2.1 Cube Set Representation Using 0-Sup-BDDs

Coudert and Madre developed a method for representing cube sets using BDDs called *Meta-products*[CMF93]. Meta-products are BDD representations for characteristic functions of cube sets. In their method, two variables are used for each input, and the two variables determine the existence of the literal and whether it is positive or negative. Coudert and Madre also presented further reduced graphs, named *Implicit Prime Sets (IPS)*[CM92], to represent prime cube sets efficiently. However, IPSs can represent only *prime* cube sets and cannot provide canonical expressions for *general* cube sets.

By using 0-Sup-BDDs, we can represent any cube set simply, efficiently and uniquely. Figure 7.1 illustrates a cube set that can be seen as a set of combinations using two variables for literals x_k and $\overline{x_k}$. x_k and $\overline{x_k}$ never appear together in the same cube. At least one should be 0. The 0's are conveniently suppressed in 0-Sup-BDDs. The number of cubes exactly equals the number of 1-paths in the graph and the total number of literals can be counted in a time proportional to the size of the graph.

The basic operations for the cube set representation based on 0-sup-BDDs are:

"0"	returns ϕ. (no cube)
"1"	returns **1**. (the tautology cube)
And0(P, var)	returns $(\overline{var} \cdot P)$.
And1(P, var)	returns $(var \cdot P)$.
Factor0(P, var)	returns the factor of P by $\overline{var}$.
Factor1(P, var)	returns the factor of P by var.
FactorX(P, var)	returns the cubes in P excluding $var, \overline{var}$.
Union(P, Q)	returns $(P + Q)$.
Intsec(P, Q)	returns $(P \cap Q)$.
Diff(P, Q)	returns $(P - Q)$.
CountCubes(P)	returns number of cubes.
CountLits(P)	returns number of literals.

"0" corresponds to the 0-terminal node on 0-sup-BDDs, and "1" corresponds to the 1-terminal node. Any one cube can be generated by applying a number of And0() and And1() to "1". The three Factor operations mean that

$$P = (\overline{var} \cdot \text{Factor0}) + (var \cdot \text{Factor1}) + \text{FactorD}.$$

Intsec() is different from logical AND operation. It only extracts the common cubes in the two cube sets. These operations are simply composed of 0-sup-BDD operations. Their execution time is almost proportional to the size of the graphs.

7.2.2 ISOP Algorithm Based on 0-Sup-BDDs

Using this new cube set representation, we have developed a program for generating prime-irredundant cube sets based on the *ISOP algorithm*, described in Chapter 5. Our program converts a conventional BDD representing a given Boolean function into a 0-sup-BDD representing a prime-irredundant cube set.

The algorithm is summarized in this expansion as:

Table 7.1: Generation of prime-irredundant cube sets.

Name	#BDD	#Cube	#Literal	#ZBDD	Time(s)
add8	41	2,519	23,211	88	0.5
add16	81	655,287	11,468,595	176	1.3
mult6	1,358	2,284	22,273	3,315	5.7
mult8	10,766	35,483	474,488	45,484	82.3
achil8	24	8	32	24	0.3
achil8n	24	6,561	59,049	24	0.3
c432	27,302	84,235	969,028	14,407	83.2
c499	52,369	348,219,564	6,462,057,445	195,356	2400.1
c880	19,580	114,299	1,986,014	18,108	78.7
c1908	17,129	56,323,472	1,647,240,617	233,775	385.6
c5315	32,488	137,336,131	742,606,419	41,662	886.4

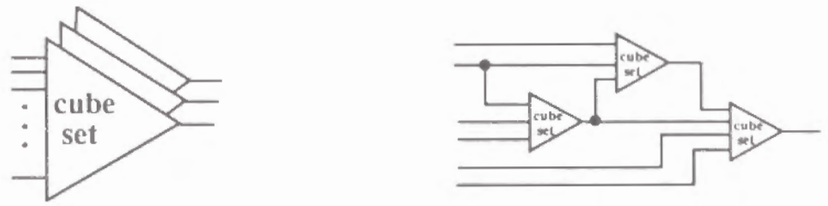
$$isop = \overline{v} \cdot isop_0 + v \cdot isop_1 + isop_d$$

where *isop* represents the prime-irredundant cube set, and v is one of the input variables. This expansion reveals that *isop* can be divided into three subsets containing $\overline{v}$, v, and the others. When $\overline{v}$ and v are excluded from each cube, the three subsets of $isop_1, isop_0$ and $isop_d$ should also be prime-irredundant. Based on this expansion, the algorithm recursively generates a prime-irredundant cube set.

We found that the ISOP algorithm can be accelerated through the use of the new cube set representation based on 0-sup-BDDs. We prepared a hash-based cache to store the results of each recursive call. Each entry in the cache is formed as pair (f, s), where f is the pointer to a given BDD and s is the pointer to the result of 0-sup-BDD. On each recursive call, we check the cache to see whether the same sub-function f has already appeared, and if so, we can avoid duplicate processing and return result s directly. By using this technique, we can execute the ISOP algorithm in a time almost proportional to the size of graph, independent of number of the cubes and literals.

We implemented the program on a SPARC work station and generated prime-irredundant cube sets from the functions of large-scale combinational circuits. The results are shown in Table 7.1. The circuits used for this experiment were: an 8+8 bit adder (add8), a 16+16 bit adder (add16), a 6×6 bit multiplier (mult6), a 8×8 bit multiplier (mult8) and a 24-input *Achilles' heel function*[BHMSV84] (achil8p) and its complement (achil8n). Other items were chosen from benchmarks of MCNC'90.

Column *#BDD* indicates the size of the initial BDDs. *#Cube* and *#Literal* indicate the scale of the generated prime-irredundant cube sets as well as the memory requirement, provided we use a classical representation, such as a linear linked list. The actual memory requirements are shown in the column *#ZBDD*.



(a) Initial cube sets. (b) Final cube set network.

Figure 7.2: Factorization of cube sets.

The results show that extremely large prime-irredundant cube sets containing billions of literals can now be easily generated. This has never been practical before. Noteworthy is *c5315*, where only 5.5 literals appear in a cube on an average, while the function has 178 inputs (i. e., 356 literals). In this case combinations are quite sparse, and as a result, 0-sup-BDDs can reduce the memory requirement dramatically. In general, reduced cube sets consist of very sparse cubes and use of 0-Sup-BDDs is effective. When we manipulate cube sets using intermediate variables to represent multi-level logic, cube sets are sparser, and 0-Sup-BDDs even more effective.

7.3 Factorization of Implicit Cube Set Representation

In this section, we present a fast algorithm for factorizing implicit cube representation using 0-sup-BDDs. We demonstrate a new multi-level logic optimizer that we have developed based on the fast factorization method.

7.3.1 Weak-Division Method

In general, two-level logics can be factorized into more compact multi-level logics. The initial two-level logics are represented with large cube sets for primary output functions, as shown in Fig. 7.2(a). When we determine a good intermediate logic, we make a cube set for the intermediate logic and reduce the other existing cube sets by using a new intermediate variable. Eventually, we construct a multi-level logic which consists of small size of cube sets as illustrated in Fig 7.2(b). The multi-level logic consists of hundreds of cube sets, each of which is very small. On the average, less than 10 variables out of hundreds are used for each cube set. They yield so sparse combinations that the use of 0-sup-BDDs is quite effective. Another benefit of 0-sup-BDDs is that we do not have to fix the number of variables beforehand. We can use additional variables whenever an intermediate logic is found.

Weak-division (or algebraic division) is the most successful and prevalent method for generating multi-level logics from cube sets.

For example, cube set expression

$$f = a\ b\ d + a\ b\ \overline{e} + a\ b\ \overline{g} + c\ d + c\ \overline{e} + c\ h$$

can be divided by $(a\ b+c)$. By using an intermediate variable p, the expression can be rewritten as:

$$f = p\ d + p\ \overline{e} + a\ b\ \overline{g} + c\ h \quad p = a\ b + c.$$

In the next step, f will be divided by $(d + \overline{e})$ in a similar manner.

Weak-division does not exploit all of Boolean properties of the expression and is only an algebraic method. In terms of result quality, it is not as effective as other stronger optimizing methods, such as *the transduction method*[MKLC87]. However, weak-division is still important because it is used for generating initial logic circuits for other strong optimizers, and applied to large-scale logics that cannot be handled by strong optimizers.

The conventional weak-division algorithm is executed by computing the common part of quotients for respective cubes in the divisor. For example, suppose the two expressions are

$$\begin{aligned} f &= a\ b\ d + a\ b\ \overline{e} + a\ b\ \overline{g} + c\ d + c\ \overline{e} + c\ h, \\ p &= a\ b + c. \end{aligned}$$

f can be rewritten as:

$$f = a\ b\ (d + \overline{e} + \overline{g}) + c\ (d + \overline{e} + h).$$

The quotient (f/p) can then be computed as:

$$\begin{aligned} (f/p) &= (f/\ (a\ b)\ ) \cap (f/c) \\ &= (d + \overline{e} + \overline{g}) \cap (d + \overline{e} + h) \\ &= d + \overline{e} \end{aligned}$$

The remainder $(f\%p)$ is computed using the quotient as:

$$\begin{aligned} (f\%p) &= f - p\ (f/p) \\ &= a\ b\ \overline{g} + c\ h. \end{aligned}$$

Using the quotient and the remainder, f is reduced as:

$$\begin{aligned} f &= p\ (f/p) + (f\%p) \\ &= p\ \overline{d} + p\ e + a\ b\ \overline{g} + c\ h. \end{aligned}$$

One step in the factorization process is then completed.

The conventional weak-division algorithm requires an execution time that depends on the length of expressions (or the number of literals in f and p). This is because we have to compute a number of quotients for all cubes in the divisor. This method is impracticable when we deal with very large cube sets such as parity functions and adders. In the next section, we will present a much faster weak-division algorithm based on the implicit cube set representation using 0-sup-BDDs.


```
procedure(f/p)
{   if (p = 1) return f ;
    if (f = 0 or f = 1) return 0 ;
    if (f = p) return 1 ;
    q ← cache("f/p") ; if (q exists) return q ;
    v ← p.top ; /* the highest variable in p */
    (f0, f1, fd) ← factors of f by v ;
    (p0, p1, pd) ← factors of p by v ;
    q ← p ;
    if (p0 ≠ 0) q ← f0/p0 ;
    if (q = 0) return 0 ;
    if (p1 ≠ 0)
        if (q = p) q ← f1/p1 ;
        else q ← q ∩ (f1/p1) ;
    if (q = 0) return 0 ;
    if (pd ≠ 0)
        if (q = p) q ← fd/pd ;
        else q ← q ∩ (fd/pd) ;
    cache("f/p") ← q ;
    return q ;
}
```

Figure 7.3: Implicit weak-division algorithm.

7.3.2 Fast Weak-Division Algorithm Based on 0-Sup-BDDs

Our method generates (f/p) from f and $p(\neq 0)$ in the implicit cube set representation. The algorithm is described in Fig. 7.3. The basic idea here is that we do not compute quotients for respective cubes in the divisor, but rather for sub-cube sets factored by an input variable. Here, v is the highest-ordered input variable contained in p, and cube sets f and p are factored into three parts as:

$$\begin{aligned} f &= \overline{v}\ f_0 + v\ f_1 + f_d, \\ p &= \overline{v}\ p_0 + v\ p_1 + p_d. \end{aligned}$$

The quotient (f/p) can then be written as:

$$(f/p) = (f_0/p_0) \cap (f_1/p_1) \cap (f_d/p_d).$$

Each sub-quotient term can be computed recursively. The procedure is eventually broken down into trivial problems and the results are obtained. If one of the values for p_0, p_1 and p_d is zero, we may skip the term. For example, if $p_1 = 0$, then $(f/p) = (f_0/p_0) \cap (f_d/p_d)$. Whenever we find that one of the values for (f_0/p_0), (f_1/p_1) and (f_d/p_d) becomes zero, $(f/p) = 0$ becomes obvious and we no longer need to compute.

```
procedure(f · g)
{   if (f.top < g.top) return (g · f) ;
    if (g = 0) return 0 ;
    if (g = 1) return f ;
    h ← cache("f · g") ;  if (h exists) return h ;
    v ← f.top ; /* the highest variable in f */
    (f0, f1, fd) ← factors of f by v ;
    (g0, g1, gd) ← factors of g by v ;
    h ← v̄(f0 · g0 + f0 · gd + fd · g0)
      +v(f1 · g1 + f1 · gd + fd · g1) + fd · gd ;
    cache("f · g") ← h ;
    return h ;
}
```

Figure 7.4: Implicit multiplication algorithm.

In this algorithm, the example shown in the previous section is computed as:

$$\begin{aligned}
& (a\,b\,d + a\,b\,\overline{e} + a\,b\,\overline{g} + c\,d + c\,\overline{e} + c\,h)/(a\,b + c) \\
= \;& (b\,d + b\,\overline{e} + b\,\overline{g})/b \;\cap\; (c\,d + c\,\overline{e} + c\,h)/c \\
= \;& (d + \overline{e} + \overline{g})/1 \;\cap\; (d + \overline{e} + h)/1 \\
= \;& (d + \overline{e} + \overline{g}) \cap (d + \overline{e} + h) \\
= \;& d + \overline{e}.
\end{aligned}$$

In the same way as that for the ISOP algorithm, we prepared a hash-based cache to store results for each recursive call and avoid duplicate execution. Using the cache technique, we can execute this algorithm in a time almost proportional to the size of the graph, regardless of the number of cubes and literals.

In order to obtain the remainder of division $(f\%p) = f - p\,(f/p)$, we need to compute the algebraic multiplication between two cube sets. This procedure can also be described recursively and executed quickly using the cache technique, as illustrated in Fig. 7.4.

7.3.3 Divisor Extraction

For multi-level logic synthesis based on the weak-division method, the quality of results greatly depends on the choice of divisors. *Kernel extraction*[BSVW87] is the most common and sophisticated method. This method extracts good divisors and has been used successfully in practical systems such as MIS. However, this method is complicated and time consuming for very large cube sets. We need a simple and fast method for finding divisors in implicit cube sets.

The basic algorithm is described as follows.


```
Divisor(f)
{   v ← a literal appears twice in f ;
    if(v exist) return Divisor(f/v) ;
    else return f ;
}
```

If there is a literal which appears more than once in a cube set, we compute the factor for the literal. Repeating this recursively, we eventually obtain a divisor, which is the same as the one called *level-0 kernel* in the kernel extraction method used in MIS. With this method, factors for a literal can be computed quickly in the implicit representation. Whether a literal appears more than once can be checked efficiently by looking on the branch of the graph.

A different divisor may be obtained for another order of factoring literals. When two or more possible literals are located, we first choose a literal which is defined later so that the extracted divisor may have variables nearer to the primary inputs. This rule allows us to maintain a shallow depth of the circuits.

Use of a common divisor for multiple cube sets may yield better results, but locating common divisors is complicated and time consuming for large cube sets. So far, we have only been able to extract single output divisors and apply them to all the other cube sets. If there is a cube set providing non-zero quotient for the divisor, we force to execute the division. At least one cube set and sometimes more can be divided by a common cube.

Using the complement function for the divisor, we sometimes can attain more compact expressions. For example,

$$f = a\,\overline{c} + b\,\overline{c} + \overline{a}\,\overline{b}\,c$$

can be factorized using a complement divisor as:

$$\begin{aligned}
f &= p\,\overline{c} + \overline{p}\,c, \\
p &= a + b.
\end{aligned}$$

It is not easy to compute the complement function in the cube set representation. We transform the cube set into a conventional BDD for the Boolean function of the divisor, and make a complement for the BDD. We then regenerate a cube set from the inverted BDD using the ISOP algorithm. This strategy seems as if it would require a large computation time, however, the actual execution time is comparatively small in the entire process because the divisors are always small cube sets.

7.4 Implementation and Experimental Results

Based on the above method, we implemented a multi-level logic synthesizer. The basic flow of the program is illustrated in Fig. 7.5. Starting with non-optimized multi-level logics, we first generate BDDs for the Boolean functions of primary outputs under a heuristic ordering of input variables[MIY90]. Next, we

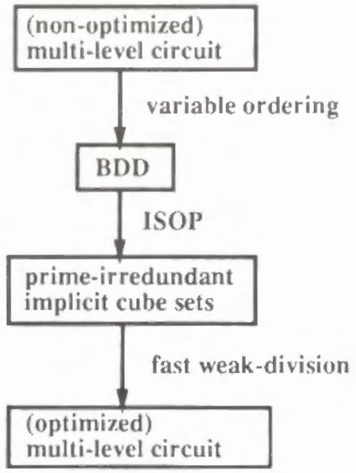


Figure 7.5: Basic flow of multi-level logic synthesizer.

transform the BDDs into prime-irredundant implicit cube sets using the ISOP algorithm. The cube sets are then factorized into optimized multi- level logics using the fast weak-division method.

We wrote the program with C++ language on a SPARC station 2. We compared our experimental results with those of the MIS-II using a conventional cube based method. The Boolean optimization commands in MIS-II were not used. We used algebraic method only to evaluate the effects of our implicit weak-division method. When cube sets become too large, MIS-II provides an option to optimize multi-level logics without going through two-level logics, however, we forced to flatten them into cube sets in this experiment.

The results are shown in Table 7.2. The circuits were an 8 bit and 16 bit parity functions (xor8, xor16), a 16+16 bit adder (add16), 6×6 bit multiplier (mult6), and other items are chosen from MCNC'90 benchmarks. The column *Time* indicates the total time of execution. We can now quickly flatten and factorize circuits, even for parity functions and adders, which have never been practicable with conventional methods. The results reveals that our method is much faster than MIS-II, and that difference is remarkable when factorizing large cube sets. The number of literals in the optimized logic networks were almost same as those revealed by the conventional method in MIS-II.

7.5 Conclusion

We have developed a fast weak-division method for implicit cube sets based on 0-sup-BDDs. Computation time of this method is almost proportional to the size of 0-sup-BDDs, and is independent of the number of cubes and literals in cube

Table 7.2: Results of multi-level logic synthesis.

Name	Two-Level Logic		Our Method		MIS-II	
	#Literal	#ZBDD	#Literal	Time(s)	#Literal	Time(s)
xor8	1,152	28	28	0.3	28	38.3
xor16	557,056	60	60	0.7	-	(>10h)
add16	11,468,595	176	257	6.9	-	(>10h)
mult6	22,273	3,315	6,802	2,900.7	-	(>10h)
9sym	1,036	42	117	1.8	83	29.8
vg2	914	102	102	1.7	97	33.9
alu4	5,539	1,129	1,148	64.5	1,319	3,751.6
apex1	4,115	1,768	2,521	209.6	2,863	10,945.1
apex2	15,530	1,144	253	29.5	-	(>10h)
apex3	4,679	1,539	2,221	158.2	2,132	1,926.6
apex4	8,055	1,545	3,473	462.4	3,509	1,345.9
apex5	7,603	2,387	1,185	58.7	1,206	156.9
c432	969,028	14,407	1,510	692.3		(>10h)

sets. Experimental results indicates that we can quickly flatten and factorize circuits, even for parity functions and adders, which have never been practicable before. Our method greatly accelerates logic synthesis systems and enlarges the scale of applicable circuits.

There are still some room to improve the results. Thus, we have adopted an easy strategy for choosing divisors, but more sophisticated strategies may be possible. Moreover, Boolean division method for implicit cube sets is worth investigating for this purpose.

Chapter 8

Arithmetic Boolean Expressions

8.1 Introduction

In the research and development of digital systems, Boolean expressions are sometimes used to handle problems and procedures. It is a cumbersome job to calculate Boolean expressions by hand, even if they have only a few variables. If they have more than five or six variables, we might as well give up. This problem motivated us to develop a Boolean Expression Manipulator (BEM)[MIY89], which is an interpreter that uses BDDs to calculate Boolean expressions. It enables us to check the equivalence and implications of Boolean expressions quite easily. It has helped us in developing VLSI design systems and solving combinatorial problems.

Although most discrete problems can be described by Boolean expressions, arithmetic operators, such as addition, subtraction, multiplication and comparison, are convenient for describing many practical problems, as seen in 0-1 linear programming. Such expressions can be rewritten using logic operators only, but this can make them complicated and hard to read. In many cases, arithmetic operators provide simple problem descriptions of problems.

In this chapter, we present a new Boolean Expression Manipulator, which we call BEM-II, that allows the use of arithmetic operators. BEM-II can directly solve problems represented by a set of equalities and inequalities, which are dealt with in 0-1 linear programming. Of course, it can also manipulate ordinary Boolean expressions as well. We developed several output formats for displaying expressions containing arithmetic operators.

In the following sections, we first show a method for manipulating Boolean expressions with arithmetic operations. We then present implementation of the arithmetic Boolean expression manipulator and its applications.

8.2 Manipulation of Arithmetic Boolean Expressions

As we discussed above, although most discrete problems can be described by using Boolean expressions, arithmetic operators are useful for describing many practical problems. For example, a majority function with five inputs can be expressed concisely by using arithmetic operators

$$x_1 + x_2 + x_3 + x_4 + x_5 \geq 3.$$

Using only Boolean expressions, this function become complicated as:

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge x_2 \wedge x_5)$$
$$\vee (x_1 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_5) \vee (x_1 \wedge x_4 \wedge x_5)$$
$$\vee (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_4 \wedge x_5)$$
$$\vee (x_3 \wedge x_4 \wedge x_5).$$

In this section, we describe an efficient method that uses BDDs to represent and manipulate expressions with arithmetic operators.

8.2.1 Definitions

For manipulating Boolean expressions that include arithmetic operators, we define *arithmetic Boolean expressions* and *Boolean-to-integer functions*, which are extended models of conventional Boolean expressions and Boolean functions.

Definition 8.1 Arithmetic Boolean expressions *are extended Boolean expressions which contain not only logic operators, but also arithmetic operators, such as addition (+), subtraction (−), and multiplication (×). Any integer number is allowed to be used as a constant term in the expression, but input variables are restricted to either 0 or 1. Equality (=) and inequalities ($<, >, \leq, \geq, \neq$) are defined as operations which return a value of either 1 (true) or 0 (false).*

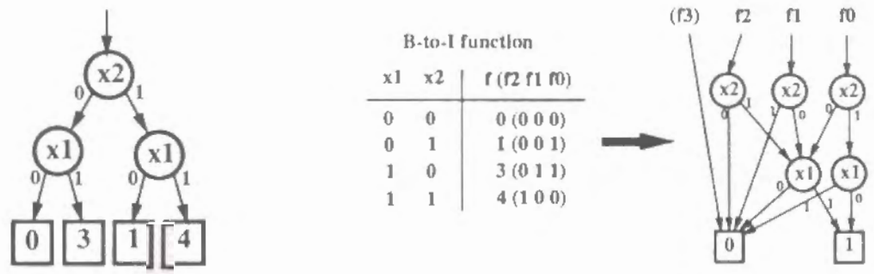
For example, $(3 \times x_1 + x_2)$ is an arithmetic Boolean expression with respect to the variables $x_1, x_2 \in \{0, 1\}$. $(3 \times x_1 + x_2 < 4)$ is another example.

When ordinary logic operations are applied to integer values other than 0 and 1, we define them as bit-wise logic operations for binary-coded numbers, like in many programming languages. For example, $(3 \vee 5)$ returns 7. Under this modeling scheme, conventional Boolean expressions become special cases of arithmetic Boolean functions.

The value of the expression $(3 \times x_1 + x_2)$ becomes 0 when $x_1 = x_2 = 0$, or 4 when $x_1 = x_2 = 1$. We can see that an arithmetic Boolean expression represents a function from binary-vector to integer: $(B^n \rightarrow I)$. We call this function a **Boolean-to-integer (B-to-I) function**, which has been discussed in Section 4.2. The operators in arithmetic Boolean expressions are defined as operations on B-to-I functions. We can calculate B-to-I functions for arithmetic

	x_1x_2 00	01	10	11
$3 \times x_1$	0	0	3	3
$3 \times x_1 + x_2$	0	1	3	4
$3 \times x_1 + x_2 < 4$	1	1	1	0

Figure 8.1: Computation of arithmetic Boolean expressions.



(a) A MTBDD. (b) A BDD vector.

Figure 8.2: Representation for B-to-I functions.

Boolean expressions by applying operations on B-to-I functions according to the structure of the expressions.

The procedure for obtaining the B-to-I function for the expression $(3 \times x_1 + x_2 < 4)$ is shown in Fig. 8.1. First, multiply the constant function 3 times input function x_1 to obtain the B-to-I function for $(3 \times x_1)$. Then add x_2 to obtain the function for $(3 \times x_1 + x_2)$. Finally we can get a B-to-I function for the entire expression $(3 \times x_1 + x_2 < 4)$ by applying the comparison operator $(<)$ to the constant function 4. We find that this arithmetic Boolean expression is equivalent to the expression $(\overline{x_1} \vee \overline{x_2})$.

8.2.2 Representation of B-to-I Functions

Figure 8.1 showed how a B-to-I function can be obtained by enumerating the output values for all possible combinations of the input values. This is impractical when there are many input variables since the number of combinations grows exponentially. We thus need a more efficient way to represent B-to-I functions.

As discussed in Section 4.2, there are two ways to represent B-to-I functions using BDDs: multi-terminal BDDs (MTBDDs) and BDD vectors. MTBDDs are extended BDDs with multiple terminals, each of which has an integer value (Fig. 8.2(a)). BDD vectors are the way to represent B-to-I functions with a number of BDDs by encoding the integer numbers into binary vectors (Fig. 8.2(b)).

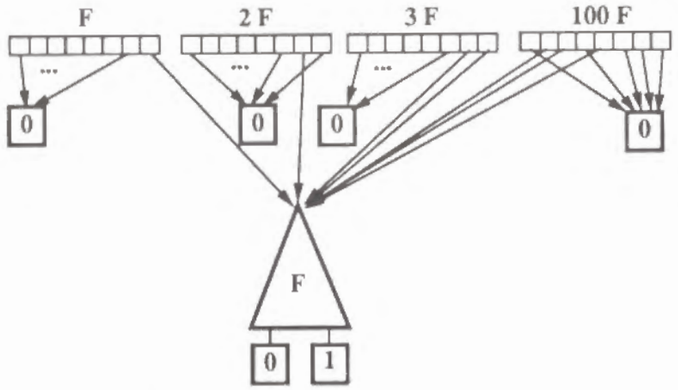


Figure 8.3: BDD vectors for arithmetic Boolean expressions.

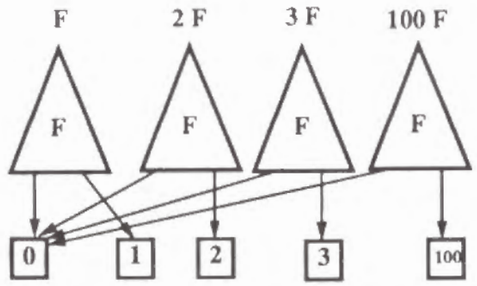


Figure 8.4: MTBDDs for arithmetic Boolean expressions.

The efficiency of the two representations depends on the nature of the objective functions. In manipulating arithmetic Boolean expressions, we often generate B-to-I functions from Boolean functions, as when we calculate $F \times 2, F \times 5$, or $F \times 100$ from a certain Boolean function F. In such cases, the BDD vectors can be conveniently shared with each other (Fig. 8.3). However, multi-terminal BDDs cannot be shared (Fig. 8.4). We therefore use BDD vectors for manipulating arithmetic Boolean expressions.

For negative numbers, we use 2's complement representation in our implementation. The most significant bit is used for the sign bit, whose BDD indicates under which conditions the B-to-I function produces a negative value. This encoding scheme requires fixing the bit length in advance, and it limits the range of numbers. We allocate a sufficiently long bit length to avoid inconvenience from this constraint.

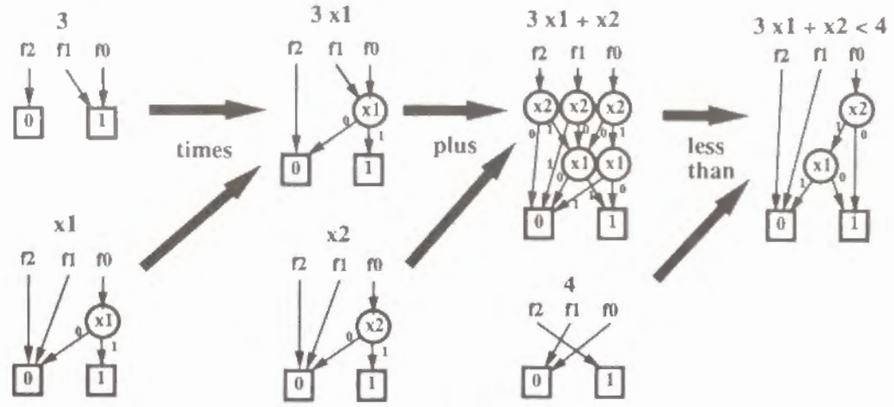


Figure 8.5: Generation of BDD vectors for arithmetic Boolean expressions.

8.2.3 Handling B-to-I functions

This section explains how to handle B-to-I functions represented by BDD vectors.

Logic operations, such as AND, OR, and EXOR, are implemented as bit-wise operations between two BDD vectors. Applying BDD operations to their respective bits, a new B-to-I function is generated. We define two kinds of inversion operations: bit-wise inversion and logical inversion. Logical inversion returns 1 only for 0, otherwise it returns 0.

Arithmetic addition can be composed using logic operations on BDDs by simulating a conventional hardware algorithm of full-adders which are designed as combinational circuits. We use a simple algorithm for a ripple carry adder, which computes from the lower bit to the higher bit, propagating carries. Other arithmetic operations, such as subtraction, multiplication, division and shifting can be composed in the same way. Exception handling should be considered for overflow and division by zero.

Positive/negative checking is immediately indicated by the sign-bit BDD. Using subtraction and sign checking, we can compose comparison operations between two B-to-I functions. These operations generate a new B-to-I function that returns a value of either 1 or 0 to express whether the equality or inequality is satisfied.

It is useful if we can find the upper (or lower) bound value of a B-to-I function for all possible combinations of input values. This can be done efficiently by using binary search. To find the upper bound, we first check whether the function can ever exceed 2^n. If there is a case in which it does, we then compare it with $2^n + 2^{n-1}$, otherwise with only 2^{n-1}. In this way, all the bits can be determined from the highest to the lowest, and eventually the upper bound is obtained. The lower bound is found in the same way.

f = 2 a + 3 b - 4 c + d

ab \ cd	00	01	11	10
00	0	1	-3	-4
01	3	4	0	-1
11	5	6	2	1
10	2	3	-1	-2

Figure 8.6: An integer Karnaugh map.

Computing the upper (or lower) bound is a unary operation for B-to-I functions; it returns a constant B-to-I function and can be used conveniently in arithmetic Boolean expressions. For example, the expression:

$$UpperBound(F) == F \quad (F \text{ is an arithmetic Boolean expression})$$

gives a function which returns 1 for the inputs that maximize F, otherwise it returns 0. Namely it computes the condition for maximizing F.

An example of calculating arithmetic Boolean expressions using BDD vectors is shown in Fig. 8.5.

8.2.4 Display Formats for B-to-I Functions

We propose several formats for displaying B-to-I functions represented by BDDs.

Integer Karnaugh maps

A conventional Karnaugh map displays a Boolean function using a matrix of logic values (0, 1). We extended the Karnaugh map to use integer numbers for each element (Fig. 8.6). We call this an *integer Karnaugh map*. It is useful for observing the behavior of B-to-I functions. Like ordinary Karnaugh maps, they are practical only for fewer than five or six input functions. For a larger number of inputs, we can make an integer Karnaugh map with respect to only six input variables, by displaying the upper (or lower) bound for the rest of variables on each element of the map.

Bit-wise expressions

When the objective function is too complicated for an integer Karnaugh map, the function can be displayed by listing Boolean expressions for respective bits of the BDD vector in the sum-of-products format. Figure 8.7 shows a bit-wise expression for the same function shown in Fig .8.6. We used the ISOP algorithm,

$$f = 2 \times a + 3 \times b - 4 \times c + d$$

$$\begin{aligned}
\pm &: (\overline{a} \wedge c \wedge \overline{d}) \vee (\overline{b} \wedge c) \\
f_2 &: (a \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge c \wedge \overline{d}) \vee (b \wedge \overline{c} \wedge d) \vee (\overline{b} \wedge c) \\
f_1 &: (a \wedge \overline{b}) \vee (a \wedge d) \vee (\overline{a} \wedge b \wedge \overline{d}) \\
f_0 &: (b \wedge \overline{d}) \vee (\overline{b} \wedge d)
\end{aligned}$$

Figure 8.7: A bit-wise expression

$$\begin{aligned}
6 &: a \wedge b \wedge \overline{c} \wedge d \\
5 &: a \wedge b \wedge \overline{c} \wedge \overline{d} \\
4 &: \overline{a} \wedge b \wedge \overline{c} \wedge d \\
3 &: (a \wedge \overline{b} \wedge \overline{c} \wedge d) \vee (\overline{a} \wedge b \wedge \overline{c} \wedge \overline{d}) \\
2 &: (a \wedge b \wedge c \wedge d) \vee (a \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \\
1 &: (a \wedge b \wedge c \wedge \overline{d}) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge d) \\
0 &: (\overline{a} \wedge b \wedge c \wedge d) \vee (\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \\
-1 &: (a \wedge \overline{b} \wedge c \wedge d) \vee (\overline{a} \wedge b \wedge c \wedge \overline{d}) \\
-2 &: a \wedge \overline{b} \wedge c \wedge \overline{d} \\
-3 &: \overline{a} \wedge \overline{b} \wedge c \wedge d \\
-4 &: \overline{a} \wedge \overline{b} \wedge c \wedge \overline{d}
\end{aligned}$$

Figure 8.8: Case enumeration format.

which has been described in Chapter 5, to generate a compact sum-of-products format on each bit. Bit-wise expression is not so helpful for showing the behavior of B-to-I functions, but it does allow us to observe the appearance frequency of an input variable and it can estimate a kind of complexity of the functions. If a function never has negative values, we can suppress the expression for the sign bit. If some higher bits are always zero, we can omit showing them. With this zero suppression, a bit-wise expression becomes a simple Boolean expression if the function returns only 1 or 0.

Case enumeration

Using case enumeration, we can list all possible values of a function and display the condition for each case using a sum-of-products format (Fig. 8.8). This format is effective when there are many input variables but the range of output values is limited.

Arithmetic sum-of-products format

It would be useful if we could display a B-to-I function as an expression using arithmetic operators. There is a trivial way of generating such an expression by using the case enumeration format. When the case enumeration method gives the values $v_1, v_2, \ldots, v_m$ and their conditions $f_1, f_2, \ldots, f_m$, we can create the expression $(v_1 \times f_1 + v_2 \times f_2 + \ldots + v_n \times f_m)$.

Using this method, $(2 \times a + 3 \times b - 4 \times c + d)$ would be displayed as:

$$\begin{aligned} & 6 \times a\ b\ \overline{c}\ d + 5 \times a\ b\ \overline{c}\ \overline{d} + 4 \times \overline{a}\ b\ \overline{c}\ d + 3 \times (a\ \overline{b}\ \overline{c}\ d + \overline{a}\ b\ \overline{c}\ \overline{d}) \\ & + 2 \times (a\ b\ c\ d + a\ \overline{b}\ \overline{c}\ \overline{d}) + (a\ b\ c\ \overline{d} + \overline{a}\ \overline{b}\ \overline{c}\ d) - (a\ \overline{b}\ c\ d + \overline{a}\ b\ c\ \overline{d}) \\ & - 2 \times a\ \overline{b}\ c\ d - 3 \times \overline{a}\ \overline{b}\ c\ d - 4 \times \overline{a}\ \overline{b}\ c\ \overline{d}. \end{aligned}$$

This expression seems too complicated compared to the original one, which has a linear form. Here we propose a method for eliminating the negative literals from the above expression and making an arithmetic sum-of-products expression which consists of arithmetic addition, subtraction, and multiplication operators only. Our method is based on the following expansion:

$$\begin{aligned} F &= x \times F_1 + \overline{x} \times F_0 \\ &= x \times (F_1 - F_0) + F_0, \end{aligned}$$

where F is the objective function, and F_0 and F_1 are sub-functions obtained by assigning 0 and 1 to input variable x. By recursively applying this expansion to all the input variables we can generate an arithmetic sum-of-products expression containing no negative literals. We can thereby extract a linear expression from a B-to-I function if it is possible. For example, the B-to-I function for $2 \times (a + 3 \times b) - 4 \times (a + b)$ can be displayed in a reduced format as $(-2 \times a + 2 \times b)$.

The arithmetic sum-of-products format seems unsuitable for representing ordinary Boolean functions. For example, $(a \wedge \overline{b}) \vee (c \wedge \overline{d})$ becomes $-a\ b\ c\ d + a\ b\ c - a\ b + a\ c\ d - a\ c + a - c\ d + c$. This expression is more difficult to read than the original one.

8.3 Applications

Using the techniques described above, we developed an arithmetic Boolean expression manipulator. This program, called BEM-II, is an interpreter with a lexical and syntax parser for calculating arithmetic Boolean expressions and displaying the results in various formats. This section gives the specifications for BEM-II and discusses some applications.

8.3.1 BEM-II Specification

BEM-II has a C-shell-like interface, both for interactive execution from the keyboard and for batch jobs from a script file. The program is written in yacc, C, and C++ languages. It runs on 32-bit UNIX machines.

Table 8.1: Operator line up in BEM-II.

(The upper operators are executed prior to the lower ones.)

( )
!(logical) ~(bit-wise) + -(unary)
* /(quotient) %(remainder)
+ -(binary)
<< >> (bit-wise shift)
< <= > >= == != (relation)
& (bit-wise AND)
^ (bit-wise EXOR)
| (bit-wise OR)
?: (if-then-else)
UpperBound( ) LowerBound( )

In BEM-II scripts, we can use two kind of variables, *input variables* and *register variables*. Input variables, denoted by strings starting with a lower-case letter, represent the inputs of the functions to be computed. They are assumed to have a value of either 1 or 0. Register variables, denoted by strings starting with an upper-case letter, are used to identify the memory to which a B-to-I function to be saved temporarily. We can describe multi-level expressions using these two types of variables, for example:

$$F = a + b\ ; \quad G = F + c.$$

Calculation results are displayed as expressions with input variables only, not using register variables. BEM-II allows 65,535 different input variables to be used. There is no limit on the number of register variables.

BEM-II supports such logical operators such as AND, OR, EXOR, and NOT, and such arithmetic operators as plus, minus, product, division, shift, equality, inequality, and upper/lower bound. The syntax for expressions generally conforms to C language specifications. *If-then-else* expressions can also be supported as $A\ ?\ B : C$, which is equivalent to

$$(\ (A \neq 0) \times B) + (\ (A = 0) \times C).$$

BEM-II parses the script only from left to right. Neither branches nor loop controls are supported. The list of available operators is shown in Table 8.1.

BEM-II generates BDD vectors of B-to-I functions for given arithmetic Boolean expressions. Since BEM-II can generate huge BDDs with millions of nodes, limited only by memory size, we can manipulate large-scale and complicated expressions. It is enough to calculate expressions that used to be manipulated by hand, of course. The results can be displayed in the various formats presented in earlier sections.

In Fig. 8.9, we show an example of the script for a *subset sum problem*, which seeks the maximum cost under an upper bound. Using BEM-II, we can generate

```
% bemII
***** Arithmetic Boolean Expression Manipulator (Ver. 4.2) *****
> symbol a b c d
> F = 2*a + 3*b - 4*c + d
> print /map F
 a b : c d
     |    00    01    11    10
  00 |     0     1    -3    -4
  01 |     3     4     0    -1
  11 |     5     6     2     1
  10 |     2     3    -1    -2
> print /bit F
 +-: !a & c & !d | !b & c
  2: a & b & !c | !a & c & !d | b & !c & d | !b & c
  1: a & !b | a & d | !a & b & !d
  0: b & !d | !b & d
> print F > 0
 a & b | a & !c | b & !c | !c & d
> M = UpperBound(F)
> print M
     6
> print F == M
 a & b & !c & d
> C = (F >= -1) & (F < 4)
> print C
 a & c & d | !a & !c & !d | b & c | !b & !c
> print /map C
 a b : c d
     |    00    01    11    10
  00 |     1     1     0     0
  01 |     1     0     1     1
  11 |     0     0     1     1
  10 |     1     1     1     0
> quit
%
```

Figure 8.9: An example of executing BEM-II.

BDDs for constraint functions of combinatorial problems specified by arithmetic Boolean expressions. This enables us to solve 0-1 linear programming problems by handling equalities and inequalities directly, without coding complicated procedures in a programming language. BEM-II can also solve problems which are expressed by non-linear expressions. BEM-II features its customizability. We can compose scripts for various applications much more easily than developing and tuning a specific program.

Here we show the application of BEM-II to several practical problems.

8.3.2 Timing Analysis for Logic Circuits

For designing high-speed digital systems, timing analysis of logic circuits is important. The orthodox approach is to traverse the circuit to find the active

Table 8.2: Results of timing analysis.

Circuit	In	Out	Gate	Number of BDD nodes	
				Timing data	Logic data
cm138a	6	8	29	235	129
sel8	12	2	43	926	268
alu2	10	6	434	16,883	4,076
alu4	14	8	809	97,318	9,326
alupla	25	5	114	22,659	2,889
mult6	12	12	411	57,777	9,496
too_large	39	3	1044	730,076	10,789
C432	36	7	255	1,689,576	10,827

path with the topologically maximum length. Takahara[Tak93] proposed a new timing analysis method using BEM-II. This method calculates B-to-I functions representing the delay time with respect to the values of the primary inputs. Using this method, we can completely analyze the timing behavior of a circuit for any combination of input values.

The B-to-I functions for the delay time can be described by a number of arithmetic Boolean expressions, each of which specifies the signal propagation on each gate. For example, on a two-input AND gate with delay D, where T_a and T_b are the signal arrival times at the two input pins, and V_a and V_b are their final logic values, the signal arrival time at output pin T_c is expressed as:

$$
\begin{aligned}
T_c &= T_b + D, \text{ when } (T_a < T_b) \text{ and } (V_a = 1),\\
T_c &= T_a + D, \text{ when } (T_a < T_b) \text{ and } (V_a = 0),\\
T_c &= T_a + D, \text{ when } (T_a > T_b) \text{ and } (V_b = 1),\\
T_c &= T_b + D, \text{ when } (T_a > T_b) \text{ and } (V_b = 0).
\end{aligned}
$$

These rules can be described by an arithmetic Boolean expression as

$$T_c = D + (\ (T_a > T_b)\ ?\ \ (V_b\ ?\ \ T_a : T_b) : (V_a\ ?\ \ T_b : T_a)\ ).$$

By calculating such expressions for all the gates in the circuit, we can generate BDD vectors for the B-to-I functions of the delay time. Table 8.2[Tak93] shows the experimental results for practical benchmark circuits. The size of the BDDs for the delay time is about 20 times less than that of the BDDs for the Boolean functions of the circuits.

The generated BDDs maintain the timing information for all of the internal nets in the circuit. Utilizing BEM-II, we can then analyze the circuits in various ways. For example, we can easily compare the delay times between two nets in the circuit.

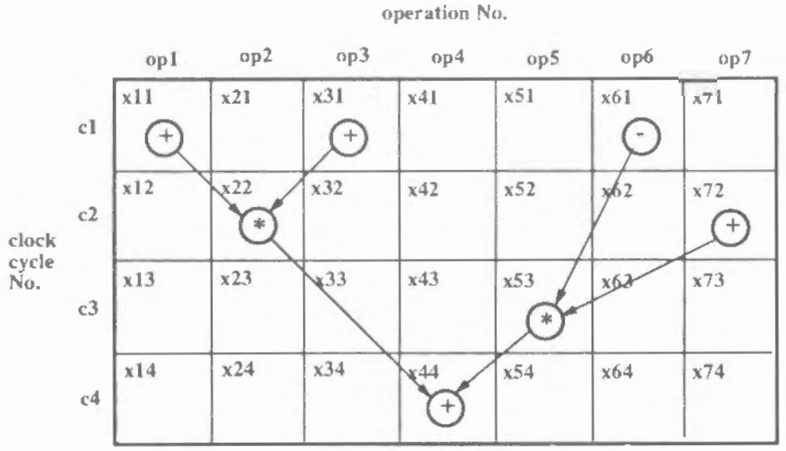


Figure 8.10: An example of data-flow graph.

8.3.3 Scheduling Problem in Data Path Synthesis

Scheduling is one of the most important subtasks that must be solved to perform data path synthesis. Miyazaki[Miy92] proposed a method for solving scheduling problems using BEM-II. The problem is to find the minimum cost scheduling for a procedure specified by a data-flow graph under such constraints as the number of operation units and the maximum clock cycles (Fig. 8.10). While this scheduling problem can be solved by using linear programming, BEM-II can also be utilized.

Assume m is the total number of operations that appear in the data-flow graph, and n is the maximum number of clock cycles. We then allocate $m \times n$ input variables from x_{11} to x_{mn}, where x_{ij} represents the i-th operation executed on the j-th clock cycle. Using this variable coding, the constraints of scheduling problem can be represented as follows.

1. Each operation has to executed once:
   $x_{11} + x_{12} + \ldots + x_{1n} = 1$
   $x_{21} + x_{22} + \ldots + x_{2n} = 1$
   $\ldots$
   $x_{m1} + x_{m2} + \ldots + x_{mn} = 1$

2. The same kind of operations cannot be executed simultaneously beyond the number of operation units. For example, when there are two adders, and the a-th, b-th, and c-th operations require an adder:
   $x_{a1} + x_{b1} + x_{c1} \leq 2$
   $x_{a2} + x_{b2} + x_{c2} \leq 2$

   $\ldots$
   $x_{an} + x_{bn} + x_{cn} \leq 2$.

3. If two operations have a dependency in the data-flow graph, the operation in the upper stream has to be executed before the one in the lower stream.
   Let $C_1 = 1 \times x_{11} + 2 \times x_{12} + \ldots + n \times x_{1n}$.
   Let $C_2 = 1 \times x_{21} + 2 \times x_{22} + \ldots + n \times x_{2n}$.
   $\ldots$
   Let $C_m = 1 \times x_{m1} + 2 \times x_{m2} + \ldots + n \times x_{mn}$.
   Then, $(C_i < C_j)$ is the condition that the i-th operation is executed before j-th one.

The logical product of all the above constraint expressions becomes the solution to the scheduling problem. Using BEM-II, we can easily specify the cost of operation and the other constraints. BEM-II analyzes the above expressions and tries to generate BDDs that represent the solutions. If it succeeds in generating BDDs in main memory, we can immediately find a solution to the problem and count the number of solutions. Otherwise it may abort. The constraint functions for many benchmark problems can be represented as feasible-size BDDs[Miy92].

8.3.4 Other Combinatorial Problems

We can utilize BEM-II for many other combinatorial problems. The 8-Queens problem is one example of a problem that can easily be described by arithmetic Boolean expressions and be solved by BEM-II.

We first allocate 64 input variables corresponding to the squares on a chessboard. These represent whether or not there is a queen on that square. The constraints that the input variables should satisfy are expressed as follows:

- The sum of eight variables in the same column is 1.
- The sum of eight variables in the same row is 1.
- The sum of variables on the same diagonal line is less than 2.

These constraints can be described with simple arithmetic Boolean expressions as:

$$\begin{aligned} Cond_1 &= (x_{11} + x_{12} + x_{13} + \ldots + x_{18} == 1) \\ Cond_2 &= (x_{21} + x_{22} + x_{23} + \ldots + x_{28} == 1) \\ &\ldots \\ Solutions &= Cond_1 \wedge Cond_2 \wedge \ldots. \end{aligned}$$

BEM-II analyzes the above expressions directly. This is much easier than creating a specific program in a programming language. The script for the 8-Queens problem took only ten minutes to create.

Table 8.3: Results for N-queens problems.

N	#Variable	#BDD	#Solution	Time(s)
8	64	2450	92	6.1
9	81	9556	352	18.3
10	100	25944	724	68.8
11	121	94821	2680	1081.9

Table 8.3 shows the results when we applied this method to the N-Queens problems. In our experiments, we solved the problem up to $N = 11$. When seeking only one solution, we can solve the problem for a larger N by using a conventional algorithm based on backtracking. However, the conventional method does not enumerate all the solutions nor count the number of solutions for larger Ns. The BDD-based method generates all the solutions simultaneously and keeps them in a BDD. Therefore, if an additional constraint is appended later, we can revise the script quite easily, without rewriting the program from the beginning. This customizability makes BEM-II very efficient in terms of the total time for programming and execution.

BEM-II can also be utilized for minimum-tree problems, traveling salesman problems, magic squares, crypt-arithmetic problems, etc. While it is second to well-optimized heuristic algorithms for solving large-scale problems, it is a useful tool for researching and developing VLSI CAD systems.

8.4 Conclusion

We have developed an arithmetic Boolean expression manipulator (BEM-II) that can easily solve many kind of combinatorial problems. BEM-II can directly analyze the equalities and inequalities in the constraints and costs of the problem, and generates BDDs that represent the solutions. It is therefore not necessary to write a specific program to solve the problem in a programming language. The customizability of BEM-II makes it very efficient in terms of total time for programming and execution. We expect it to be a useful tool for researching and developing various digital systems.

Chapter 9

Conclusions

In this thesis, we have discussed the techniques related to BDDs and their applications for VLSI CAD systems.

In Chapter 2, we presented basic algorithms of Boolean function manipulation using BDDs. We then described implementation techniques to make BDD manipulators applicable to practical problems. As an improvement of BDDs, we proposed the use of *attributed edges*, which are the edges attached with several sorts of attributes representing a certain operation. Especially, the negative edges are now commonly used because of their remarkable advantage. Using these techniques, we implemented a BDD subroutine package for Boolean function manipulation. It can efficiently represent and manipulate very large-scale BDDs containing more than million of nodes. These techniques have been developed and improved in many laboratories in the world[Bry86, MIY90, MB88, BRB90], and some program packages are opened to public. Using the BDD packages, a number of works are in progress on the VLSI CAD and other various areas in computer science.

The algorithms of BDDs are based on the quick search of the hash tables and the linked list data structure. Both of the two techniques greatly benefit from the property of the *random access machine model*, such that any data on the main memory can be accessed in a constant time. As most of computers are designed in this model, we can conclude that the BDD manipulation algorithms are fairly sophisticated and adapted to the conventional computer model.

In Chapter 3, We have discussed the properties on variable ordering, and shown two heuristic methods: DWA method and minimum-width method. The former one finds an appropriate order before generating BDDs. It refers topological information of the Boolean expression or logic circuit which specifies the sequence of logic operations. Experimental results show that the DWA method finds a tolerable order in a short computation time for many practical circuits. On the other hand, the minimum-width method finds an appropriate order for a given BDD using no additional information. It seeks a good order with a global view, not based on incremental search. In many cases, this method gives better results than the DWA method in a longer but still reasonable computation time.

The techniques of variable ordering are intensively researched still now. Un-

fortunately, it is almost impossible to have an ultimate method of variable ordering to always find best order in a practical time. We will make do with some heuristic methods according to the applications.

In Chapter 4, We discussed the representation of multi-valued functions. In many applications, we sometimes use ternary-valued functions containing don't cares. We have shown two method of handling *don't care*; ternary-valued BDDs and BDD pairs, and compared the two by introducing the D-variable. The technique of handling *don't care* are basic and important for Boolean function manipulation in many problems.

Based on the consideration of *don't care*, we extended the argument into B-to-I functions, and presented two methods; MTBDDs and BDD vectors. They can be compared by introducing bit-selection variables, as well as on the ternary-valued functions. Multi-valued logic manipulation is important to broaden the scope of BDD application. Presently, a number of researches are in progress. These techniques are useful not only for VLSI CAD but also for various areas in computer science.

In Chapter 5, we discussed the topic how efficiently transform BDD representation into other data structures. We presented ISOP algorithm for generating prime-irredundant cube sets directly from given BDDs, in contrast to the conventional cube set reduction algorithms, which temporarily manipulate redundant cube sets or truth tables. The experimental results demonstrate that our method is efficient in terms of time and space. In practical time, we can generate cube sets consisting of more than 1,000,000 literals from multi-level logic circuits which have never previously been flattened into two-level logics. Our method is more than 10 times faster than conventional methods in large-scale examples. It gives quasi-minimum numbers of cubes and literals. In terms of size of the result, the ISOP algorithm may give somewhat larger results than ESPRESSO, but there are many applications in which such an increase is tolerable. Our method can be utilized to transform BDDs into compact cube sets or to flatten multi-level circuits into two-level circuits.

In Chapter 6, we have proposed *Zero-Suppressed BDDs (0-Sup-BDDs)*, which are BDDs based on a new reduction rule. As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating *sets of combinations* to deal with many problems, not only in the digital system design but also various areas in computer science. 0-sup-BDDs can manipulate sets of combinations more efficiently than using conventional BDDs. This data structure is adapted for the sets of combinations. The effect of 0-Sup-BDDs is remarkable especially when manipulating sparse combinations. We discussed the properties of 0-sup-BDDs and their efficiency based on a statistical experiment. We then presented the basic operators for 0-sup-BDDs. Those operators are defined as the operations on sets of combinations, which slightly differ from the Boolean function manipulation based on conventional BDDs.

Based on the 0-sup-BDD techniques, we discussed the calculation of unate cube set algebra. We developed efficient algorithms for computing unate cube

set operations including multiplication and division, and show some practical applications. For solving some types of combinatorial problems, unate cube set algebra is more useful than using conventional logic operations. We expect the unate cube set calculator to be utilized as a helpful tool in researching and developing VLSI CAD systems and other various applications.

In Chapter 7, an application for VLSI logic synthesis was presented. We proposed a fast factorization method for cube set representation based on 0-sup-BDDs. Our new algorithm can be executed in a time almost proportional to the size of 0-sup-BDDs, which are usually much smaller than the number of literals in the cube set. By using this method, we can quickly generate multi-level logics from implicit cube sets even for parity functions and full-adders, which have never been possible with the conventional methods. We implemented a new multi-level logic synthesizer, and experimental results indicate our method is much faster than conventional methods and differences are more significant for larger-scale problems. Our method greatly accelerates multi-level logic synthesis systems and enlarges the scale of applicable circuits. There are still some room to improve the results. Thus, we have adopted an easy strategy for choosing divisors, but more sophisticated strategies may be possible. Moreover, Boolean division method for implicit cube sets is worth investigating for this purpose.

In Chapter 8, we presented a helpful tool for the research on computer science. We have developed an arithmetic Boolean expression manipulator (BEM-II) that can easily solve many kind of combinatorial problems. Our product features that it calculates not only binary logic operation but also arithmetic operations on multi-valued logics, such as addition, subtraction, multiplication, division, equality and inequality. Such arithmetic operations provide simple descriptions for various problems. BEM-II feeds and computes the problems represented by a set of equalities and inequalities, which are dealt with using 0-1 linear programming. It is therefore not necessary to write a specific program to solve the problem in a programming language. We discussed the data structure and algorithms for the arithmetic operations. Finally we presented the specification of BEM-II and some application examples, such as the 8-Queens problem. Experimental results indicate that the customizability of BEM-II makes it very efficient in terms of total time for programming and execution. We expect it to be a useful tool for researching and developing various digital systems.

Bibliography

[Ake78] S. B. Akers. Binary decision diagram. *IEEE Trans. on Computers*, Vol. C-27, No. 6, pp. 509–516, June 1978.

[BC94] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical report, Carnegie Mellon University, May 1994.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 46–51, June 1990.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits. In *Proc. of 1985 IEEE International Symposium Circuit and Systems (ISCAS'85), Special Session on ATPG and Fault Simulation*, June 1985.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 188–191, November 1993.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers USA, 1984.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 40–45, June 1990.

[BRKM91] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proc. of 28th ACM/IEEE Design Automation Conference (DAC'91)*, pp. 417–420, June 1991.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677–691, August 1986.

[Bry91] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, Vol. C-40, No. 2, pp. 205–213, February 1991.

[BSVW87] R. K. Brayton, R. R. A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 6, pp. 1062 1081, November 1987.

[CB89] K. Cho and R. E. Bryant. Test pattern generation for sequential MOS circuits by symbolic fault simulation. In *Proc. of 26th ACM/IEEE Design Automation Conference (DAC'89)*, pp. 418–423, June 1989.

[CHJ+90] H. Cho, G. D. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-90)*, pp. 134–137, November 1990.

[CM92] O. Coudert and J. C. Madre. A new graph based prime computation technique. In T. Sasao, editor, *New Trends in Logic Synthesis*, chapter 2, pp. 33–57. Kluwer Academic Publishers USA, 1992.

[CMF93] O. Coudert, J. C. Madre, and H. Fraisse. A new viewpoint of two-level logic optimization. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 625–630, June 1993.

[CMZ+93] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 54–60, June 1993.

[dG86] A. J. de Geus. Logic synthesis and optimization benchmarks for the 1986 DAC. In *Proc. of 23rd ACM/IEEE Design Automation Conference (DAC'86)*, pp. 78, June 1986.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvement of Boolean comparison method based on binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-88)*, pp. 2–5, November 1988.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proc. of IEEE The European Conference on Design Automation (EDAC'91)*, pp. 50–54, February 1991.

[FS87] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proc. of 24th ACM/IEEE Design Automation Conference (DAC'87)*, pp. 348 356, June 1987.

[HCO74] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A heuristic approach for logic minimization. *IBM Journal of Research and Development*, Vol. 18, No. 5, pp. 443–458, 1974.

[HMPS94] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic analysis of large finite state machines. In *Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94)*, pp. 270–275, June 1994.

[Ish92] N. Ishiura. Synthesis of multi-level logic circuits from binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan)*, pp. 74–83, March 1992.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-91)*, pp. 472–475, November 1991.

[IY90] N. Ishiura and S. Yajima. A class of logic functions expressible by a polynomial-size binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'90, Japan)*, pp. 48–54, October 1990.

[IYY87] N. Ishiura, H. Yasuura, and S. Yajima. High-speed logic simulation on vector processors. *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 3, pp. 305–321, May 1987.

[JT92] R. P. Jacobi and A. M. Trullemans. Generating prime and irredundant covers for binary decision diagrams. In *Proc. of IEEE The European Conference on Design Automation (EDAC'92)*, pp. 104 108, March 1992.

[LCM89] P. Lammens, L. J. Claesen, and H. D. Man. Tautology checking benchmarks: Results with TC. In *Proc. of IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 600 604, November 1989.

[LL92] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Trans. on Computers*, Vol. C-41, No. 6, pp. 661–664, June 1992.

[LPV93] Y.-T. Lai, M. Pedram, and S. B. Vrudhula. FGILP: An integer linear program solver based on function graphs. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 685–689, November 1993.

[LS90] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-90)*, pp. 88–91, November 1990.

[MB88] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proc. of 25th ACM/IEEE Design Automation Conference (DAC'88)*, pp. 205–210, June 1988.

[MF89] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-89)*, pp. 556–559, November 1989.

[Min90] S. Minato. Shared binary decision diagrams for efficient Boolean function manipulation. Master's thesis, Department of Information Science, Faculty of Engineering, Kyoto University, February 1990.

[Min93a] S. Minato. BEM-II:an arithmetic Boolean expression manipulator using BDDs. *IEICE Trans. Fundamentals*, Vol. E76-A, No. 10, pp. 1721–1729, October 1993.

[Min93b] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 272–277, June 1993.

[MIY89] S. Minato, N. Ishiura, and S. Yajima. Symbolic simulation using shared binary decision diagram. In *Record of the 1989 IEICE Fall Conference (in Japanese)*, pp. 1.206–207, SA–7–5, September 1989.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 52–57, June 1990.

[Miy92] T. Miyazaki. Boolean-based formulation for data path synthesis. In *Proc. of IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS'92)*, pp. 201–205, December 1992.

[MKLC87] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method—design of logic networks based on permissible functions. *IEEE Trans. on Computers*, Vol. C-38, No. 10, pp. 1404–1424, June 1987.

[Mor70] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Trans. on Computers*, Vol. C-19, No. 6, pp. 504–509, June 1970.

[MSB93] P. McGeer, J. Sanghavi, and R. K. Brayton. Espresso-signature: A new exact minimizer for logic functions. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 618–624, June 1993.

[MWBSV88] S. Malik, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-88)*, pp. 6–9, November 1988.

[Oku94] H. G. Okuno. Reducing combinatorial explosions in solving search-type combinatorial problems with binary decision diagram. *Trans. of Information Processing Society of Japan (IPSJ), (in Japanese)*, Vol. 35, No. 5, pp. 739–753, May 1994.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, pp. 42–47, November 1993.

[Tak93] A. Takahara. A timing analysis method for logic circuits. In *Record of the 1993 IEICE Spring Conference (in Japanese)*, pp. 1.120, A–20, March 1993.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering of a shared binary decision diagram. Technical Report 93-6, Department of Information Science, Faculty of Science, University of Tokyo, December 1993.

[TIY91] N. Takahashi, N. Ishiura, and S. Yajima. Fault simulation for multiple faults using BDD representation of fault sets. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-91)*, pp. 550–553, November 1991.

Acknowledgment

I would like to express my sincere appreciation to Professor Shuzo Yajima of Kyoto University for his continuous guidance, interesting suggestions, accurate criticisms and encouragements during this research.

I would also like to express my thanks to Dr. Nagisa Ishiura of Osaka University who introduced me to the research field of VLSI CAD and BDDs, and has been giving me invaluable suggestions, accurate criticisms and encouragements throughout this research.

I also acknowledge interesting comments that I have received from Professor Hiromi Hiraishi of Kyoto Sangyo University, Professor Hiroto Yasuura of Kyushuu University, and Associate Professor Naofumi Takagi of Nagoya University.

I would like to thank Dr. Kiyoharu Hamaguch, Mr. Koich Yasuoka and Mr. Shoichi Hirose of Kyoto University, Associate Professor Hiroyuki Ochi of Hiroshima City University, Mr. Yasuhito Koumura of Sanyo Corporation and Kazuya Ioki of IBM Japan Corporation for their interesting discussions and collaborations in implementing a BDD program package in Chapter 2.

I am indebted to Professor Randal E. Bryant of Carnegie Mellon University who introduced BDD techniques into the VLSI CAD. I started this research with his papers and received helpful comments from him.

I would also like to thank Dr. Masahiro Fujita and Mr. Yusuke Matsunaga of Fujitsu Corporation, who started to work on BDDs in early time. I referred their papers many times, and sometimes had fruitful discussions with them.

I am grateful to Professor Saburo Muroga of University of Illinois at Urbana-Champaign and Professor Tsutomu Sasao of Kyushu Institute of Technology for their instructive comments and discussions on the techniques of logic synthesis.

I also wish to thank Dr. Olivier Coudert of DEC Paris Research Laboratory for his discussions on the implicit set representation, which led to the idea of Zero-suppressed BDDs in Chapter 6.

I would like to thank Professor Gary D. Hachtel and Associate Professor Fabio Somenzi of University of Colorado at Boulder for their fruitful discussions on the application of BDDs for combinatorial problems in Chapter 8.

I am also grateful Professor Robert K. Brayton, Dr. Patrik McGeer, Dr. Yoshinori Watanabe and Mr. Yuji Kukimoto of University of Calfornia at Berkeley for their interesting discussions and suggestions on the techniques of logic synthesis based on BDDs in Chapter 5 and 7.

Thanks are also due to my colleagues in NTT Research Laboratories. I gratefully acknowledge the collaboration with Dr. Toshiaki Miyazaki and Dr. Atsushi Takahara in developing and evaluating the BEM-II program in Chapter 8. I am also grateful to Mr. Hiroshi G. Okuno and Mr. Masayuki Yanagiya for their interests in the BEM-II applications. I also thank Mr. Noriyuki Takahashi for his helpful comments on the application of 0-sup-BDDs for fault simulation in Chapter 6.

I express special gratitude to Mr. Yasuyoshi Sakai, Dr. Osamu Karatsu, Mr. Tamio Hoshino, Mr. Makoto Endo, and all the other members of NTT Advanced LSI Laboratory for giving me an opportunity to write this thesis, and for their helpful advices and encouragements.

Thanks are due to all the members of the Professor Yajima's Laboratory, especially to Mr. Yasuhiko Takenaga, Mr. Hiroyuki Ogino and Mr. Kazuyoshi Takagi, for their discussions and helpful supports throughout this research.

Lastly, I thank my parents and my wife for their patience, support and encouragement.

List of Publications by the Author

Major Publications

1. S. Minato, N. Ishiura, and S. Yajima. Fast tautology checking using shared BDD. In *Proc. of IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 107–111, November 1989.

2. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. of 27th ACM/IEEE Design Automation Conference (DAC'90)*, pp. 52–57, June 1990.

3. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams for efficient Boolean function manipulation. *Trans. of Information Processing Society of Japan (IPSJ), (in Japanese)*, Vol. 32, No. 1, pp. 77–85, January 1991.

4. S. Minato. Minimum-width method of variable ordering for binary decision diagrams. *IEICE Trans. Fundamentals*, Vol. E75-A, No. 3, pp. 392–399, March 1992.

5. S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan)*, pp. 64–73, April 1992.

6. S. Minato. Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE Trans. Fundamentals*, Vol. E76-A, No. 6, pp. 967–973, June 1993.

7. S. Minato. BEM-II: An arithmetic Boolean expression manipulator using BDDs. *IEICE Trans. Fundamentals*, Vol. E76-A, No. 10, pp. 1721–1729, October 1993.

8. S. Minato. Fast weak-division method for implicit cube representation. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI'93, Japan)*, pp. 423–432, October 1993.

9. S. Minato. Techniques for BDD manipulation on computers. *The Journal of Information Processing Society of Japan (IPSJ), (in Japanese)*, Vol. 34, No. 5, pp. 593–599, May 1993.

10. S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 272–277, June 1993.

11. S. Minato. Calculation of unate cube set algebra using zero-suppressed BDDs. In *Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94)*, pp. 420–424, June 1994.

Technical Reports

1. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram for efficient Boolean function manipulation. *Technical Report of IEICE on VLSI Design Technologies, (in Japanese)*, Vol. 89, No. 338, pp. 39–46, VLD89-80, December 1989.

2. S. Minato. Fast generation of irredundant sum-of-products from binary decision diagrams. *Technical Report of IEICE on VLSI Design Technologies, (in Japanese)*, Vol. 91, No. 376, pp. 25–32, VLD91-107, December 1991.

3. S. Minato. Minimum-width method of variable ordering for binary decision diagram reduction. In *Proc. of IEICE The 4th KARUIZAWA Workshop on Circuits and Systems, (in Japanese)*, pp. 271–276, April 1991.

4. S. Minato. Techniques of Boolean function manipulation based on binary decision diagrams. In *Proc. of IEICE The 5th KARUIZAWA Workshop on Circuits and Systems (in Japanese)*, pp. 161–166, April 1992.

5. S. Minato. BEM-II: An arithmetic Boolean expression manipulator using binary decision diagrams. *Technical Report of IEICE on Computation, (in Japanese)*, Vol. 92, No. 447, pp. 25–32, COMP92-75, January 1993.

6. S. Minato. Fast weak-division method for implicit cube sets using zero-suppressed BDDs. *Technical Report of IEICE on VLSI Design Technologies, (in Japanese)*, Vol. 93, No. 504, pp. 55–62, VLD93-200, March 1994.

Convention Records

1. S. Minato, N. Ishiura, and S. Yajima. Symbolic simulator using truth table representation. In *Record of 37th IPSJ National Convention, (in Japanese)*, pp. 1757–1758, 3U-2, September 1988.

2. S. Minato, N. Ishiura, and S. Yajima. On Boolean function manipulation using shared binary decision diagram. In *Record of 38th IPSJ National Convention, (in Japanese)*, pp. 1371–1372, 5S-9, March 1989.

3. S. Minato, N. Ishiura, and S. Yajima. Symbolic simulation using shared binary decision diagram. In *Record of the 1989 IEICE Fall Conference, (in Japanese)*, pp. 1.206–1.207, SA-7-5, September 1989.

4. S. Minato, N. Ishiura, and S. Yajima. Variable shifter for shared binary decision diagram. In *Record of the 1990 IEICE Spring Conference, (in Japanese)*, pp. 1.115, A-115, March 1990.

5. S. Minato. Minimum-width method of variable ordering for shared binary decision diagrams. In *Record of 42nd IPSJ National Convention, (in Japanese)*, pp. 6.158, 2J-5, March 1991.

6. S. Minato. Fast generation of irredundant sum-of-products using binary decision diagrams. In *Record of 43rd IPSJ National Convention, (in Japanese)*, pp. 6.179, 2R-10, October 1991.

7. S. Minato. Variable shifter for shared binary decision diagram. In *Record of the 1992 IEICE Fall Conference, (in Japanese)*, pp. 1.51, A 51, September 1992.

8. S. Minato. Zero-suppressed BDDs and their applications. In *Record of the 1993 IEICE Spring Conference, (in Japanese)*, pp. 1.368–1.369, SA-2-2, March 1993.

9. S. Minato. Complemental set operations for zero-suppressed bdds. In *Record of the 1994 IEICE Spring Conference, (in Japanese)*, pp. 1.141, A-140, March 1994.